OPTIMISTIC SEMANTIC SYNCHRONIZATION

A Thesis
Presented to
The Academic Faculty

by

Jaswanth Sreeram

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2011

OPTIMISTIC SEMANTIC SYNCHRONIZATION

Approved by:

Professor Santosh Pande, Advisor
College of Computing
Georgia Institute of Technology

Professor Hyesoon Kim
College of Computing
Georgia Institute of Technology

Professor Joel Saltz
College of Computing
Georgia Institute of Technology

Professor Karsten Schwan
College of Computing
Georgia Institute of Technology

Professor Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: September 2011

To my parents Prasad and Vijaya Lakshmi

and

my brother Sushil

ACKNOWLEDGEMENTS

Being a doctoral student has been a wonderful experience and several people have contributed to making it enjoyable. First and foremost I would like to thank my advisor Dr. Santosh Pande for his excellent guidance and for his enthusiasm in finding and solving interesting research problems - a trait I greatly admire in him. I would also like to thank him for all the time, funding and significant intellectual labor that he has contributed towards my research work. I will always cherish the numerous stimulating discussions we have had over the years. I would also like to thank my thesis committee for their helpful feedback and for their insightful questions. I would especially like to thank Dr. Sudhakar Yalamanchili for giving me the opportunity to pursue graduate studies at Georgia Tech.

I am especially grateful to my fellow doctoral students Tushar Kumar and Romain Cledat for making my Ph.D experience productive as well as fun and for teaching me so many things. I would like to thank current and ex-members of my research lab Sarang Ozarde, Ashwini Bhagwat, Sangho Lee and Changhee Jung for being great people to work with.

My time at Georgia Tech was enjoyable in large part due to the wonderful friends I made here. I'd like to thank Rakshita Agarwal, Martin Levihn, Vishakha Gupta, Muralidhar Padala and Johnathan Gladin for their company and the memories.

Lastly, I would like to thank my parents Prasad and Vijaya Lakshmi and my brother Sushil for their love, support and encouragement during this long and sometimes difficult journey.

TABLE OF CONTENTS

DEDICATION . . . . . **iii**

ACKNOWLEDGEMENTS . . . . . **iv**

LIST OF TABLES . . . . . **x**

LIST OF FIGURES . . . . . **xi**

SUMMARY . . . . . **xiii**

I INTRODUCTION **1**

1.1 Related Work 4

1.1.1 Conflict Recovery 4

1.1.2 Value-aware and Relaxed Synchronization 6

1.1.3 Relaxed Synchronization and Imprecise Computation 7

1.1.4 Parallel Transactional Workloads 8

1.2 Our Approach 9

II CORRECTIVE CONFLICT RECOVERY IN MEMORY TRANSACTIONS **11**

2.1 Semantic Corrective Recovery 14

2.1.1 Specification and Semantics 14

2.1.2 Execution Model 15

2.2 Automatically Synthesized Corrective Handlers 18

2.2.1 Execution Model 19

2.3 Generating Checkpoint Operations 21

2.3.1 Persistent First-Class Continuations 22

2.3.2 Reducing State Saving Overheads 27

2.4 Runtime Support 30

2.4.1 TM Model 31

2.5 Safety 34

2.5.1 Opacity 37

2.5.2 Isolation: . . . 37
2.6 Experimental Evaluation . . . 38
2.6.1 Note on overheads . . . 46
2.7 Conclusions . . . 48
III IRREVOCABLE TRANSACTIONS VIA STATIC LOCK ASSIGNMENT . . . 50
3.1 Hybrid Optimistic-Pessimistic Concurrency . . . 52
3.1.1 Why irrevocability is important for performance . . . 52
3.2 Design . . . 54
3.2.1 *Must* and *May Access* Analysis using DSA . . . 55
3.3 Transaction Interference Graph . . . 56
3.3.1 Construction . . . 57
3.3.2 Pruning . . . 58
3.4 Lock Allocation and Assignment . . . 61
3.5 Runtime Support . . . 62
3.5.1 TM Model . . . 62
3.5.2 Access & Commit Protocol for Revocable Transactions . . . 63
3.5.3 Access & Commit Protocol for Irrevocable Transactions . . . 64
3.6 Experimental Evaluation . . . 64
3.6.1 Insights . . . 71
3.7 Conclusion . . . 76
IV VALUE-AWARE SYNCHRONIZATION . . . 77
4.0.1 Value-aware Synchronization . . . 78
4.1 Approximate Store Value Locality . . . 79
4.1.1 Approximate Value Locality in Critical Sections . . . 80
4.2 Strong False-conflicts . . . 82
4.3 Weak False-conflicts . . . 85
4.4 Specifying Imprecise Sharing . . . 86
4.4.1 Choice of Comparison Functions . . . 86

4.4.2 Thresholded Types 87
4.5 Avoiding Strong and Weak False-conflicts 89
4.5.1 Detecting Approximately-Local Stores 90
4.5.2 Avoiding Conflicts due to Approximately-Local Stores . . . 91
4.6 Experimental evaluation 93
4.6.1 Experimental Setup 93
4.6.2 Case Studies 94
4.7 Related Work 104
4.7.1 Transaction Nesting 104
4.7.2 Silent Stores, Value Locality and Reuse 104
4.7.3 Relaxed Synchronization and Imprecise Computation 105
4.8 Conclusions 105

V PARALLELIZING A REAL-TIME PHYSICS ENGINE USING SOFTWARE TRANSACTIONAL MEMORY 108

5.1 ODE Overview 110
5.1.1 Collision Detection 111
5.1.2 Dynamics Simulation 112
5.2 Parallel Transactional ODE 113
5.2.1 Global Thread Pool 114
5.2.2 Parallel Collision Detection using Spatial Decomposition . . . 114
5.2.3 Parallel Island Processing 117
5.2.4 Phase Separation 119
5.2.5 Feedback between phases 121
5.3 Issues 122
5.3.1 Conditional Synchronization 122
5.3.2 Memory management and application controlled alloc/de-alloc. 123
5.4 Experimental Evaluation 124
5.4.1 Execution time 126
5.4.2 Frame rate 126

5.4.3 Abort rate . . . 127
5.4.4 Thread utilization . . . 127
5.4.5 Transaction Read/Write Sets . . . 128
5.4.6 Scalability Optimizations . . . 129
5.5 Conclusion . . . 132
VI A RELAXED-CONSISTENCY TRANSACTION MODEL . . . 133
6.0.1 RSTM . . . 134
6.0.2 Contributions . . . 136
6.1 Relaxed consistency STM . . . 137
6.1.1 Conflict Reduction between Concurrent Transactions . . . 137
6.1.2 Coordinating Execution among Long-Running Concurrent Transactions . . . 138
6.2 RSTM Language Specification . . . 139
6.2.1 Group Consistency . . . 139
6.2.2 Progress Indicators . . . 142
6.3 Implementation . . . 144
6.3.1 Overview . . . 144
6.3.2 Zone-based management . . . 146
6.3.3 API overview . . . 147
6.3.4 Operational aspect of commits . . . 149
6.3.5 Runtime Optimizations . . . 151
6.4 Results . . . 153
6.4.1 A dynamic particle system . . . 153
6.4.2 Relaxation . . . 154
6.4.3 Experimental Evaluation . . . 155
6.5 Related work . . . 156
6.5.1 Relaxed consistency . . . 156
6.5.2 C/C++ language extension . . . 157
6.6 Conclusion . . . 157

VII CONCLUSION 159

7.1 Future Research 160

LIST OF TABLES

1 All numbers are for 4 threads. Column (A) is the **percentage of checkpoint restores** that ultimately resulted in a commit of a transaction that would have otherwise aborted. Column (B) is the **average size** in bytes of the state saved by a checkpoint operation. Column (C) is the **average call stack depth** of a checkpoint save operation, relative to the transaction's own stack frame 42

2 Reduction in number of memory references due to checkpointing. All numbers are for 8 threads. 46

3 Description of programs & input sets. †=STAMP benchmark or library [3] 65

4 Reduction in number of memory references due to *Irr*. All numbers are for 8 threads. 73

5 Read/Write set sizes 127

6 Table showing the number of Aborts, Commits, Transaction Throughput in Transactions per second and the ratio of the Transaction Throughput and the Theoretical Peak Transaction Throughput. P is the number of particles in the system and N is the number of threads. 152

LIST OF FIGURES

1 Lifetime of a memory transaction that uses *lazy-validation* and *commit-time lock acquisition* 2

2 List search 17

3 A transaction checkpoint 19

4 Saving and restoring the state of the stack on a conflict 23

5 (a) Overview of compiler pass to checkpoint transactional regions (b) routines for atomic list search 25

6 Simplified IR generated by the compiler pass in (a) for the code in (b) 26

7 A transaction-private, circular buffer with k entries for saving and retrieving ordered checkpoints 29

8 Aborts Vs. Threads in `list` 40

9 Speedup in execution time over a parallel TL2 baseline version of the program running with the same number of threads (each bar shows the ratio b_n/c_n where b_n is the wall clock execution time of the plain TL2 version of the program and c_n is the execution time of the checkpointed version). 41

10 Average number of checkpoint restores successful commit 43

11 Aborts 43

12 Overhead of checkpoint saving in an execution of `list` with very high-contention - 60%/20%/20% `find/insert/remove` and a small key range. Each of the lines shows speedup over single-threaded TL2 for a specific value of *n_freq*, the frequency of checkpointing as described in Section 3.2 47

13 Parallel Speedup from our Hybrid Irrevocability scheme over single-threaded TL2 for (a) `list` (b) `genome` 66

14 Parallel Speedup from our Hybrid Irrevocability scheme over single-threaded TL2 for (a) `kmeans` (b) `intruder` 68

15 Parallel Speedup from our Hybrid Irrevocability scheme over single-threaded TL2 for (a) `labyrinth` (b) `ssca2` 69

16 Parallel Speedup from our Hybrid Irrevocability scheme over single-threaded TL2 for (a) `vacation` (b) `yada` 70

17 Plot showing the impact of dynamic transaction size on the speedup obtained for the STAMP suite. Workloads with larger average dynamic transactions size show higher maximum speedups 72

18 Plot showing the impact of dynamic contention on the speedup obtained for the STAMP suite. Workloads with high average abort rates show higher speedups 75

19 Approximate Shared Value Similarity in Critical Sections 80

20 Example of two threads with Strong and Weak False-conflicts 84

21 Extensions to native types for specifying thresholds and comparison functions 89

22 `bayes` 96

23 kmeans 98

24 particle 103

25 ODE overview 113

26 Scene used in evaluating parallel ODE 124

27 Scalability 125

28 Aborts and Offloads 126

29 Speedup in speculative parallel island discovery relative to the single-threaded algorithm. The speculative version is *conflict-free* and *synchronization-free* in this case 131

30 Declaring Group Consistency 142

31 Monitoring Progress Indicators from other Transactions 143

32 API for the STM_Manager 148

33 API for STM_Transactions 149

34 Incremental Communication 153

SUMMARY

Within the last decade multi-core processors have become increasingly commonplace with the power and performance demands of modern real-world programs acting to accelerate this trend. The rapid advancements in designing and adoption of such architectures mean that there is a serious need for programming models that allow the development of correct parallel programs that execute efficiently on these processors. A principle problem in this regard is that of efficiently synchronizing concurrent accesses to shared memory. Traditional solutions to this problem are either inefficient but provide programmability (coarse-grained locks) or are efficient but are not composable and very hard to program and verify (fine-grained locks). Optimistic Transactional Memory systems provide many of the composability and programmability advantages of coarse-grained locks and good *theoretical* scaling but several studies have found that their performance in practice for many programs remains quite poor primarily because of the high overheads of providing safe optimism. Moreover current transactional memory models remain rigid - they are not suited for expressing some of the complex thread interactions that are prevalent in modern parallel programs. Moreover, the synchronization achieved by these transactional memory systems is at the *physical* or *memory* level.

This thesis advocates a position that memory synchronization problem for threads should be modeled and solved in terms of synchronization of underlying program values which have semantics associated with them. It presents optimistic synchronization techniques that address the *semantic* synchronization requirements of a parallel program instead.

These techniques include methods to 1) enable optimistic transactions to recover

from expensive sharing conflicts without discarding all the work made possible by the optimism 2) enable a hybrid pessimistic-optimistic form of concurrency control that lowers overheads 3) make synchronization *value-aware* and *semantics-aware* 4) enable finer grained consistency rules (than allowed by traditional optimistic TM models) therefore avoiding conflicts that do not enforce any semantic property required by the program. In addition to improving the *expressibility* of specific synchronization idioms all these techniques are also effective in improving parallel performance. This thesis formulates these techniques in terms of their purpose, the extensions to the language, the compiler as well as to the concurrency control runtime necessary to implement them. It also briefly presents an experimental evaluation of each of them on a variety of modern parallel workloads. These experiments show that these techniques significantly improve parallel performance and scalability over programs using state-of-the-art optimistic synchronization methods.

CHAPTER I

INTRODUCTION

The widespread popularity of multi-core processors has made it necessary to provide programmers with programming models that enable them to develop parallel programs that are both correct, efficient and scalable. The Transactional Memory (TM) model [4] has been widely studied and is touted as an elegant abstraction to express data synchronization. Such synchronization is expressed via specifying *atomic* blocks of code which are guaranteed to execute atomically - each atomic block of code appears to execute at once in during some indivisible instant of time. Therefore in contrast with fine-grained locks programmers using memory transactions can simply specify where atomicity is needed instead of also having to specify how to achieve it. This programmability advantage is the primary appeal of TM language extensions and systems.

Memory transactions were conceptualized from database transactions and they retain many of the traits of their database counterparts - guaranteeing ACID: strong atomicity, consistency, isolation and durability, separating atomicity from the method for achieving it and so on. However database transactions capture very different computation than modern real-world parallel programs. Such transactions typically capture the business logic of commercial or enterprise workloads where the ACID properties above are desirable. Contrast this with a modern real world parallel program such as a state-of-the art parallel game engine. It is not clear that simply using analogues of database transactions to manage data synchronization in such an application will result in good parallel performance or be programmer friendly. Real DB transactions are oriented around inserting, querying, deleting records and performing

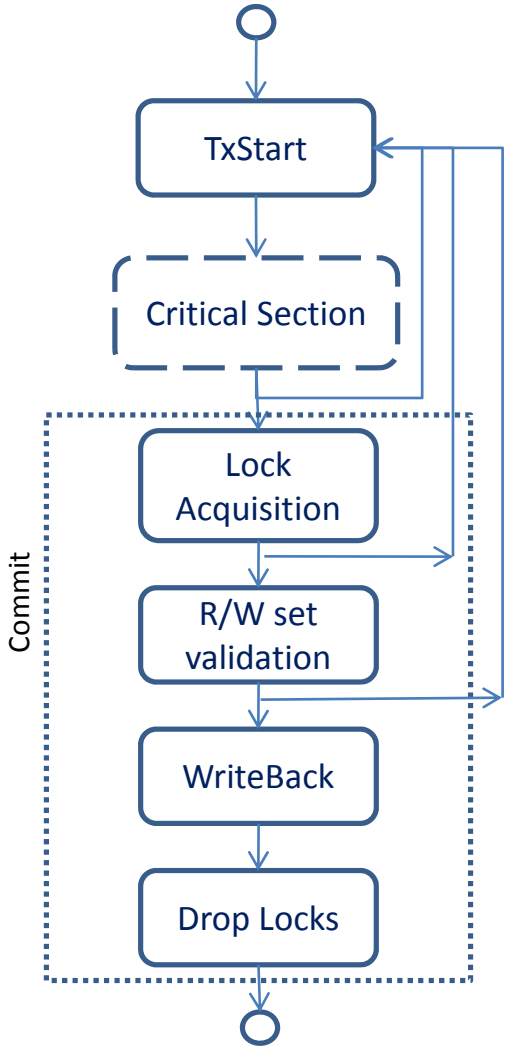


Figure 1: Lifetime of a memory transaction that uses *lazy-validation* and *commit-time lock acquisition*

some relatively simple operations on the returned data. Moreover the data schema manipulated by the user is relatively simple - tables, rows and columns. Many critical regions in modern parallel programs however implement much more complex functionality such as constraint or equation solvers, physical simulation or some non-trivial algorithm. And these critical regions often turn out to have a significant influence on overall parallel performance. Furthermore in contrast with database transactions many of these critical regions have the programmer interact with complex data structures - e.g., a scene graph or voxel-octrees in graphics and interactive simulations. The oft-used standard database example of concurrent deposits and withdrawals from a bank account may be a good simple representative case for thinking about database transactions and their properties but it does not capture the complexity and diversity of behavior in modern general purpose parallel programs.

A simple conceptual and programmatic interface for specifying atomic sections in parallel programs is certainly useful and memory transactions fit this role. However using the database notions of atomicity, consistency and isolation as the sole basis for the transactional programming model limits the diversity of synchronization idioms that can be expressed using this interface. Consider the following three properties provided by a TM system:

1. **Atomicity**: All TM systems provide the *atomicity* guarantee. In many TM systems when a transaction reads state that has been overwritten by another concurrent transaction that committed, the reader transaction is aborted and restarted. This is automatically guaranteed without regard to whether it is desirable in the context of the program's semantics. Of course, this guarantee is important for the correctness of many programs (indeed this property is extremely well aligned with the semantics desirable in common database applications) but for others it may be unnecessary and even undesirable.

2. **Isolation**: A transaction does not have any knowledge of other concurrent

transactions. In combination with the *atomicity* property above, this means that the TM model dictates that the reader transaction should abort regardless of which specific writer transaction performed the update, even if such behavior is not required by the programs semantics.

3. **No user involvement**: Some TM systems allow the user or the programmer to provide annotations or hints to turn on or off specific behaviors or algorithms in the TM runtime such as log compaction, eager or lazy conflict detection, commit-time or encounter-time locking etc. The expectation is that the programmer has the best knowledge of which of these options is most suitable for his program and will supply the annotations appropriately. However most TM systems do not allow the programmer to specify behavior such as specifying meaningful actions on important events like aborts and commits or see the transaction's state. As far as the program is concerned the state maintained by the transaction itself is off-limits. There are good reasons for this limitations, two of them being preserving programmability and preserving portability between different TM systems. So while this limitation makes it easier for novice programmers to reason about synchronization, it also severely limits what kinds of semantics other programmers can express in their transactions.

1.1 Related Work

There has been a significant amount of interest in efficient software and hardware transactional memory models and systems recently. Here we discuss the works that are relevant to this thesis.

1.1.1 Conflict Recovery

There is a substantial amount of literature on contention management for transactions and conflict resolution in particular. The studies in [14][9][1] propose various

resolution schemes which decide which of a pair of conflicting transactions is allowed to commit. However none of these allow for both transactions in the conflict to successfully commit. In [29] the authors propose a TM model that in theory allows two conflicting transactions to commit provided the online opacity-permissiveness property is preserved. The DASTM system in [23] is a dependence-aware STM in which data is forwarded between two transactions that have a dependence so that both of them can commit safely. Abstract Nested Transactions [20] allow a programmer to specify operations that are likely to be involved in benign conflicts and which can be re-executed. In [27] the authors propose annotating boosted transactions with checkpoints which allows them to partially abort. To our knowledge this work was the first to propose the notion of transaction checkpointing and this work remains the closest to the work presented in this thesis. These checkpoints were defined in the context of boosted objects with *commutative* methods and storing and saving state including the program stack and active frames was done manually. In contrast, checkpoints in our work can be placed at arbitrary points in the transaction without needing commutativity of operations and their generation and execution is completely transparent to the programmer. In [128] the authors describe an HTM protocol and system that supports intermediate state checkpointing. However, this system does not appear to perform complete checkpoints - specifically, the state of the stack is not saved - this is critical since the checkpoint may have been saved in a stack frame that has since returned (and therefore the checkpoint cannot be restored if the stack is not saved). This is a common occurrence in most of the programs we studied.

In TMs with open nesting [32] physical serializability is traded off for abstract serializability. With open nesting two transactions may conflict at the memory level but both may be permitted to execute if the abstract state of shared data is consistent with some serial execution. The RetCon [33] hardware mechanism tracks symbolic dependences between shared values and uses it to repair transactions. The Twilight

STM system [36] augments transactions with special *irrevocable* code that repairs the transactions when inconsistencies are detected before transaction commit. The Galois model in [6] model and transactional boosting in [34] rely on commutativity properties of methods and both allow for eliminating structural conflicts. Methodologies for developing self-adjusting programs - programs that are able to automatically and efficiently respond to changes in their inputs, have been studied for a few decades now. A number of algorithms in domains such as graph problems and geometry have been shown to have efficient incremental algorithms An exhaustive survey of prior work in this area is in [35]. Such programs may be specified in a special language or framework such as [2], [39] [37] that provide runtimes for recording dependences and other information that are used to direct the re-execution. Finally there is a significant amount of work on reconciling conflicting updates in mobile and distributed database systems [10] which is closely related to the present work.

1.1.2 Value-aware and Relaxed Synchronization

1.1.2.1 Transaction Nesting

The topic of open nesting in software transactional memory systems has been studied extensively [25, 26]. The main purpose of using open nesting is to separate physical conflicts from semantic conflicts since the programmer usually only cares about the latter. Therefore strict physical serializability is traded for *abstract serializability*. Abstract Nested Transactions [20] allow a programmer to specify operations that are likely to be involved in benign conflicts and which can be executed.

1.1.2.2 Silent Stores, Value Locality and Reuse

The phenomenon of silent stores has been extensively studied in the computer architecture community [22] and there have been numerous architectural optimizations suggested to exploit the same. Similarly, the phenomenon of *load value locality* has also been studied extensively [11]. Both these concepts basically establish that in

many programs, values accessed by loads and stores tend to have a repetitive nature to them. In addition, techniques based on *value prediction* exploit the locality of values loaded in a program to apply optimizations such as cache prefetching. In [21] the authors explore the phenomenon of *frequent values* - values which collectively form the majority of values in memory at an instant during program execution. In [18], the STM system uses a form of value based conflict detection for improving performance. To our knowledge, this is the only STM system that is explicitly program value-aware. In [19, 16] the authors investigate the detection and bypassing of trivial instructions for improving performance and reducing energy consumption. Frameworks such as memoization [24], function caching [37] and value reuse [41] have been proposed to allow programs to reuse intermediate results by storing results of previously executed FP instructions and matching an instruction to check if it can be bypassed by reusing a previous result.

1.1.3 Relaxed Synchronization and Imprecise Computation

The idea of relaxed consistency systems has been studied in a few contexts. Zucker studied relaxed consistency and synchronization [132] from a memory model and architectural standpoint. In [67], the authors propose a weakly consistent memory ordering model to improve performance. In [28], the authors redefine and extend isolation levels in the ANSI-SQL definitions to permit a range of concurrency control implementations. In [13] the authors propose techniques to provide improved concurrency in database transactions by sacrificing guarantees of full serializability - weak isolation was achieved by reducing the duration for which transactions held read-/write locks. A more recent work [17] work proposes Transaction Collection Classes that use multi-level transactions and open nesting, through which concurrency can be improved by relaxing isolation when full serializability is not required. In [6], the authors propose new programming constructs to improve parallelism by exploiting

the semantic commutativity of certain methods invocations.

1.1.4 Parallel Transactional Workloads

Several researchers have studied various aspects of parallelizing physics computations for applications from domains ranging from robotics, virtual environments and scientific simulations, to animation [61, 58, 64, 59]. In [64] the authors describe a voxel based parallel collision detection algorithm for distributed memory machines. This algorithm is similar to the abstract space based collision detection scheme discussed in this thesis. ParFUM [60] is a framework based on Charm++ for developing parallel applications that manipulate unstructured meshes and supports efficient collision detection. In [51] the authors study the performance of a parallel implementation of the Barnes-Hut algorithm for n-body simulation that uses octree based subdivision for computing particle interactions. In [62] the authors present an algorithm for *continuous collision detection* between deformable bodies that can be executed at interactive rates on present day multi-core machines.

Lee-TM [52] is an implementation of Lee's routing algorithm using transactional memory. While the algorithm exhibits large amount potential parallelism the transactional implementation has been shown to have modest scalability. AtomicQuake [63] is an implementation of a parallel Quake game server using transactions. The parallelization is at the level of clients connected to the server - operations for a client are performed on the server by the worker thread that the client is mapped to. Support for transactions is provided by the compiler [55] instead of a library based TM. The programs in STAMP [12] consist of a variety of parallel transactional workloads that represent pieces of larger applications and which can be executed with one of several STM or HTM systems. TMunit [54] is a framework for developing unit tests for evaluating STM systems. RMS-TM [53] is a TM benchmark suite consisting of programs and application kernels. STMBench [50] is a synthetic benchmark that that

contains transactions with widely varying characteristics and which operate on non-trivial data structures. Thus while it is very useful for finding problems with specific implementations and stretching the limits of TM designs, it is not representative of any real-world program.

1.2 Our Approach

This thesis makes the case that relaxing the *atomicity*, *isolation* and *user-involvement* properties is meaningful in some programs and that the apparent simplicity of using database style transactions does not necessarily make expressing complex semantics in some modern parallel programs easier. The following chapters describe patterns and phenomena that commonly occur in parallel programs that cannot be easily captured by the traditional notions of memory transactions in that they require some violation of the strict notions of atomicity, consistency, isolation or require user involvement. They also describe specific methods that extend transactional semantics to either express or exploit these phenomena. Briefly, this thesis makes the following technical contributions:

1. **Transaction Checkpointing & Corrective Conflict Recovery:** It proposes the notion of "corrective conflict handlers" which when used in conjunction with a novel conflict recovery scheme, enable a pair of conflicting transactions to recover from the conflict constructively by repairing their read/write sets at runtime and eventually commit. Chapter II describes the syntactic and semantic properties of these handlers and discuss automated methods to synthesis them from the original transaction.

2. **Hybrid Irrevocable Transactions via Static Lock Assignment:** Most state-of-the-art optimistic concurrency control system suffer from large execution time and memory overheads stemming from the need to continuosly track accesses to shared values. Chapter III describes compiler-driven interference

estimation techniques that when coupled with a hybrid *optimistic-pessimistic* transaction runtime model, allows multiple concurrent irrevocable transactions to execute safely along with normal optimistic transactions. We show that this type of hybrid execution model has significant performance and programmability advantages over both pure optimistic and pessimistic exeuction models.

3. **Value-aware Synchronization:** Chapter IV describes and characterizes the phenomenon of "Approximate Value Locality" in parallel programs and discusses techniques to exploit this property in programs that use optimistic concurrency control such as memory transactions. It also presents the results of characterizing the effect of these programs on program semantics particularly on the quality of results produced by the program.

4. **Parallelizing Rigid-body Physics with Transactions:**In spite of the recent interest in transactional systems, most of the studies investigating the use and optimization of these systems have been limited to smaller benchmarks and suites containing small to moderate sized programs. In Chapter V we present our experiences in using software transactions to parallelize ODE a large, commercial-grade, real-time physics engine that is widely used in hundreds of games and game engines.

5. **Relaxed Consistency Transactions:** Chapter VI outlines a form of relaxed synchronization that allows certain kinds of *physical* conflicts to be bypassed provided program semantics are not affected. It presents the notion of *consistency* groups that are collections of program values on which consistency rules are applied, instead of over all the values access in an optimistic critical section. Such relaxed synchronization when used appropriately increases transactional throughput substantially as shown in our experiments.

CHAPTER II

CORRECTIVE CONFLICT RECOVERY IN MEMORY TRANSACTIONS

In systems that implement concurrency control using memory transactions, a critical section which could potentially access the same shared data as other concurrent threads continues to execute until it detects real conflicts (either at the time of an access or later). A conflict occurs for example when a concurrent thread has written to a variable that the critical section read. When this occurs the results and intermediate values computed so far in the critical section are rendered invalid and are are therefore discarded. In other words when some (abstract) inputs to the critical section are perturbed it aborts the current computation, discards the outputs and restarts the computation.

Let T be a critical section implemented using a memory transaction. The code in T computes some function f whose inputs are the set of shared variables that T reads (the read-set R) and T's local state. The outputs of f are produced into T's write set W. If another concurrent thread writes a value to a program variable in R then T suffers a data sharing conflict. It will then discard the output it has produced into W, abort and retry from scratch. In other words, when a change is made to f's inputs (by the other thread) during f's execution it leads to a re-evaluation of f with the new inputs. This re-evaluation affects performance adversely for two reasons. Firstly there is a significant overhead associated with the set-up and tearing-down of the data structures that enable optimism (access sets, filters) in addition to deallocation/allocation of memory and other bookkeeping. Moreover any locks that may have been acquired have to be released and re-acquired when T is restarted from

scratch. Secondly, re-evaluation discards all of the state computed by the previous instance of the same computation. Therefore each re-evaluation is oblivious of the work performed in all the previous evaluations. Indeed some of this state is invalid since f's inputs were changed and this state may depend directly on these inputs. But in some cases some of this state could be reused directly if it did not depend on f's inputs at all. Finally in some cases, the intermediate state can be reused after adjusting it to account for the new inputs.

Transactions can check for conflicts at several points during their lifetime. In *lazy validation* systems conflicts are checked for every access inside the transaction. While this means large overheads are incurred, *doomed* transactions can be detected early and less work is wasted. In an *eager* system conflicts are checked at commit time and some TM systems use a hybrid approach for different types of conflicts. Regardless of the validation scheme, *conflicts discovered in most optimistically concurrent critical sections cannot be resolved without at least one abort.* For large long running critical sections or for those which have high levels of contention for shared data, this fact means that a large amount of work executed speculatively is unavoidably wasted when the critical section tries to commit. Techniques such as *transaction check-pointing* [27] , open and closed *nesting* and *abstract nested transactions* [20] have been studied which propose to lower the overhead of aborts by only partially undoing the effects of a transaction in the case of an conflict. Other systems such as DASTM [23] automatically forward values between a pair of conflicting transactions so that both may commit. Several proposals have introduced methods for *multiversion reconciliation* in mobile databases to reintegrate (often conflicting) updates to global data from multiple clients while preserving serializability[10].

This work proposes a practical mechanism for solving conflicts in which a transaction which experiences a conflict attempts to recover from the conflict by correcting its state including its read/write sets on-the-fly. This recovery action is contained

in a *handler* which is nested within the body of the transaction. A transaction that uses a handler can *"roll forward"* through a conflict and not only re-use the state it has computed so far, when implemented in TM systems that use locking, it can also retain (most of) the locks it has acquired. Using handlers does not require reasoning about properties such as commutativity or abstract inverses and does not fundamentally change transactional semantics and properties such as *opacity* are preserved. Moreover these handlers can be generated completely automatically and we discuss a few optimizations that can be used to make them efficient. These handlers can be used to realize two broad *transaction repair* mechanisms: completely restoring the transaction to some point, then re-executing it from there and making *limited, localized* corrections to the transaction's state by re-executing small portions of it.

The actions specified in corrective handlers can specify either high-level, algorithm-driven recovery actions such as ones used in Incremental Algorithms [35]. That is, the specification of the recovery action relies on some knowledge of specific semantics of the algorithm. For example, for a transaction implementing a solution to the parallel Shortest-Paths problem, might specify a handler that leverages a known incremental algorithm for handling concurrent changes to the graph being analysed. Such handlers are hard to generate automatically since they require non-trivial reasoning about the specific algorithm being implemented as we show in the case of the Shortest-Path algorithm discussed in detail below.

On the other hand, corrective recovery actions can also be low-level specifications derived from the program's structure itself and in this case they can be synthesized automatically from the program. Handlers can therefore be constructed in two ways corresponding to the two classes of corrective actions - by a programmer using simple language extensions and interface for specifying the high-level algorithm-driven recovery actions, or by the compiler using a set of program analysis to infer the low-level recovery action to be implemented in the handler. The details of each of these

methods are presented in the following sections.

2.1 Semantic Corrective Recovery

In this section we present the specification and execution semantics of language constructs for corrective handlers for *generic* memory transactions. This description is independent of the specific class or atrributes of the underlying transactional memory system - in later sections we present details of our implementation of these handlers in the context of a specific type of STM system.

Briefly a *Nested Recovery Handler* (referred to as NH or simply, handler) is specified as a *contiguous*, *block-structured* set of statements within a transaction's body and executes within the context of its containing transaction.

2.1.1 Specification and Semantics

An NH is specified and registered using the keyword `RegisterHandler` within a parent transaction as follows:

Listing 2.1: Interface for specifying a handler

```
1    atomic {
      RegisterHandler(<expr>) {
3      <handler body>
      }
5    }
```

We call the containing transaction the *parent* transaction for that handler and the handler is invoked when a conflict is detected in the parent transaction for the memory location evaluated by the expression <*expr*>. A transaction may have multiple handlers specified within it provided the pair of handlers blocks do not overlap or if they do, one of them is completely contained in the other forming a *closed* nest of handlers. The body of the handler can also be generated automatically as described later, transparently to the programmer. However this common interface serves as a basis for understanding the semantics that follow.

We introduce some notation here that is used in the rest of this discussion. The set of memory locations read, written and read-and-written by a transaction T just *before* a dynamic program point p are denoted by $\mathsf{R}_{\mathsf{T}_p}$, $\mathsf{W}_{\mathsf{T}_p}$ and $\mathsf{RW}_{\mathsf{T}_p}$ respectively (we refer to these sets together as read/write sets). The state of the local variables, heap, program stack and registers are denoted by $\mathsf{L}_{\mathsf{T}_p}$, $\mathsf{STK}_{\mathsf{T}_p}$ and $\mathsf{REG}_{\mathsf{T}_p}$ respectively. We refer to the tuple

$$\mathsf{S}_{\mathsf{T}_p} : <\mathsf{R}_{\mathsf{T}_p}, \mathsf{W}_{\mathsf{T}_p}, \mathsf{RW}_{\mathsf{T}_p}, \mathsf{L}_{\mathsf{T}_p}, \mathsf{STK}_{\mathsf{T}_p}, \mathsf{REG}_{\mathsf{T}_p}>$$

as the *state* or *execution context* of a *live* transaction T just before program point p (the subscripts T_p or p may be dropped when not necessary). A nested handler body H <*expr*> is registered with its parent transaction T when execution of T encounters the `RegisterHandler`(<*expr*>) construct and <*expr*> is evaluated. Let p denote this program point and $\mathsf{S}_{\mathsf{T}_p}$ the state of T at that point. During futher execution of T or during its validation, if a conflict is detected for the memory location <*expr*> the transaction enters the handler body H with state $\mathsf{S}_{\mathsf{T}_p}$.

2.1.2 Execution Model

An informal model of the handler's execution is as follows:

1. **Invocation**: The body of the Nested Handler is entered *if and only if* a conflict is detected for the memory location evaluated by the expression <*expr*>. The conflict may have occurred at any instant between the registering of the handler and its eventual commit. The evaluation of <*expr*> itself is performed during the parent transaction's execution and this evaluation should be *side-effect free*.

2. **Accesses**: The body of a handler can access all variables in its enclosing scope. In addition it can make transactional accesses to (new or previously accessed) shared data just like its parent transaction. These accesses are validated (during the access itself, at commit-time or both depending on the TM model) just like

the other accesses made in the transaction. The handler body can also make transactional allocation/deallocation requests for heap memory.

3. **State**: The state of the parent transaction just before the handler body is entered is described by the statements in the transaction that have been executed and which occured before the registration of the handler. The precise definition of the *state* of the transaction is captured by $\mathsf{S}_{\mathsf{T}_p}$.

4. **Completion**: After the body of the handler is executed, the parent transaction *re-enters* its validation phase where *all* the accesses made during in the transaction and any accesses made in the handler are checked for conflicts and if none are found, the transaction enters its commit phase.

2.1.2.1 Properties

Opacity: When specified inside transactions that satisfy the *Opacity* property [31], nested handlers also satisfy this property. Informally this means:

- **Atomicity**: All operations performed within a *committed* transaction and its handlers appear as if they happened at some indivisible point during an instant between the start of the transaction and its commit.

- **Aborted State**: The effects of an operation performed inside an *aborted* transaction or one of its handlers are never visible to any other transaction or its handlers.

- **Consistency**: A transaction and its handlers always observe a *consistent* state of the system.

Isolation: A nested handler only observes *consistent* state, i.e., it is guaranteed to not see any updates that have not been committed by a *live* concurrent transaction. Constructing or executing the handler does not require knowledge of either (a) other

```
1 // list->head is read-only
  atomic {
3  node_t *x = list->head;
   for(;x;) {
5   if(x->key == key)
     break;
7   x = tm_read(x->next);
   }
9 }
```

(a) Original

```
  // list->head is read-only
2 atomic {
   node_t *x = list->head;
4   for(;x;) {
     if(x->key == key)
6     break;
     RegisterHandler(x->next) {
8     x = tm_read(x->next)
      for(;x;) {
10     if(x->key == key)
        break;
12     x = tm_read(x->next);
      }
14   }
     x = tm_read(x->next);
16  }
  }
```

(b) With a nested handler

Figure 2: List search

concurrently executing transactions or (b) how the other transactions may have modified variables that caused the conflict (which invoked this handler) or (c) how many other transactions committed between the start of this transaction and the invocation of the handler.

A simple example of a transaction that performs a key lookup on a list is shown in Figure 2. The `tm_read` call performs a transactional read of the variable specified. Part (a) of the figure shows the original transaction and (b) shows the same transaction with a handler specified in lines 7-14. During execution of the transaction in (b), the handler is bound to the memory locations that are evaluated to by the `x→next` in each iteration of the loop. When a conflict occurs for a read operation on the `next` field of a particular node, the handler is executed in the same dynamic program context as that read operation and the handler resumes the lookup operation on the node pointed to by the new address in the `next` field in line 8.

2.2 Automatically Synthesized Corrective Handlers

When a long-running transaction experiences a conflict it is forced to abort thereby discarding all the work it has done so far and restart. Previous studies [3, 96] have observed that for many representative programs, between 25-95% of the work done by transactions is wasted due to aborts. At the same time, because of the simplicity and ease of use of the TM programming model transactions in modern real-world programs are becoming larger, long-running and often containing deep call chains, therefore increasing the average amount of work wasted due to an abort.

One way to reduce this wasted work is to enable a conflicting transaction to take a *recovery action* that enables the transaction to make forward progress. This recovery action could for example correct the read, write, read-and-write sets of the transaction or add/remove elements from them and ultimately help the transaction roll-forward and commit. Requiring the programmer to specify such a recovery action is impractical as it would defeat the programmability advantages that memory transactions provide. For long transactions containing deep call chains, describing these recovery actions would be cumbersome and require deep familiarity with the program. On the other hand, *automatically synthesizing* a recovery action is also challenging for several reasons. In order to *repair* the transaction's state, this synthesized recovery action would at a high-level, have to be aware of what portion of the transaction's state needs to be repaired and the specific values with which to repair the transaction's read/write sets. This is difficult as it requires not only the compiler to infer complex program-level semantics but also requires maintaining a dynamic program dependence graph (PDG) at run-time to decide which portion of the transaction's state needs to be augmented and/or modified to recover from the conflict (see [2] and references therein). Additionally the specific recovery action needed may be different depending on whether there was an execution time conflict (during transaction execution) or because of a conflict at validation time (during an commit attempt by the

transaction).

Our approach to this problem is rooted in the observation that *the transaction itself is a recovery action for every conflict that can occur in it.* Specifically, **for a conflict on any access in a dynamic instance of a transaction, if the transaction's state can be restored to a valid state at some dynamic program point just before the access, then the portion of the transaction after this point is a valid recovery action for that conflict**. Indeed an abort can simply be thought of as a checkpoint in which the program point at which the state is saved is at the very beginning of the transaction.

Our solution to this problem consists of a compiler pass that analyzes a transaction, generates checkpointing operations at the appropriate points and applies optimizations that reduce the overheads of maintaining and invoking these checkpoints and a runtime system that orchestrates the saving and restoration of all the checkpoints saved by a transaction. A generic transaction checkpoint that saves the state of a transaction after it has executed some set of statements (S1) and before it has executed another set of statements (S2) is as shown in Figure 1.

```
atomic {
 <..txn stmts (S1)..>
 CheckpointSave();
 <..txn stmts (S2)..>
}
```

Figure 3: A transaction checkpoint

2.2.1 Execution Model

An informal model of the execution of a checkpoint operation (such as the one in Figure 1 is as follows (implementation level details are discussed in later sections):

1. **Checkpoint Save**: When a transaction encounters a checkpoint *save* operation during its execution it saves its state and adds it to the transaction's *totally*

ordered set of saved checkpoints. The precise definition of the *state* of the transaction is explained in more detail later.

2. **Checkpoint Restore**: If a conflict is detected for an access to memory address *Addr*, the transaction *restores* the state of the transaction to some checkpoint that was saved before this access to *Addr* and if no such checkpoint exists the transaction simply aborts. After a successful checkpoint restore the transaction is in a *consistent* and *valid* state. That is:
   (a) It has not observed any uncommitted state from other transactions and
   (b) Its read-set $\mathsf{R}_{\mathsf{T}_p}$, write-set $\mathsf{W}_{\mathsf{T}_p}$ and read-and-write set $\mathsf{RW}_{\mathsf{T}_p}$ are valid and coherent

   After a checkpoint has been restored, the transaction begins to execute from the instruction following the "Checkpoint Save" above and with the same state that was captured then.

3. **Accesses**: After a checkpoint has been restored, the transaction continues to execute from the instruction following the checkpoint save step above. The control-flow paths and the set of transactional and non-transactional accesses that occur from that point on may be different from the previous execution - the transaction can access memory locations it has already accessed before or it can access new memory locations. These accesses are validated (during the access itself, at commit-time or both depending on the TM model) just like the other accesses made in the transaction.

4. **Opacity**: When invoked from inside transactions that satisfy the *Opacity* property [31], checkpoint handlers also satisfy this property.

5. **Isolation**: After a checkpoint restore, the transaction only observes *consistent* state, i.e., it is guaranteed to not see any updates that have not been committed by a *live* concurrent transaction. The transaction opacity, isolation and

coherence properties are discussed in more detail in Section 2.5.

6. **Completion**: When the transaction's body is finished executing after possibly several checkpoint saves and restores, it attempts to commit as normal and its entire read/write sets are validated. If this validation is successful (and in lock-based TMs, if the transaction is also able to acquire locks on all memory locations in its read-and-write and write-only sets), the transaction can commit.

Over the course of its execution a transaction may save multiple checkpoints. The set of checkpoints saved by a transaction have a *strict total ordering* - namely the order in which they were saved. This ordering is used on a conflict to decide which checkpoint to restore to, as restoring to checkpoint that was saved after this conflicting access occurred would not eliminate the conflict. As we discuss later, the checkpoint restoration mechanism attempts to restore the *latest checkpoint that occurred before the conflicting access*.

2.3 Generating Checkpoint Operations

In order to generate the checkpoint *save* and *restore* operations at compile-time and to invoke them at execution time, the principle questions that we need to answer are: (a) where should the compiler insert checkpoints for a given transaction (b) how can a runtime capture and restore the complete state of a transaction efficiently (c) how often should checkpoints be captured (d) how should the various checkpoints for a single instance of a transaction be validated and managed.

First we consider the problem of saving a transaction's state. The set of memory locations read, written and read-and-written by a transaction T just *before* a dynamic program point p are denoted by $\mathsf{R}_{\mathsf{T}_p}$, $\mathsf{W}_{\mathsf{T}_p}$ and $\mathsf{RW}_{\mathsf{T}_p}$ respectively (we refer to these sets together as read/write sets). The state of the local variables (both transactional and non-transactional), heap, program stack and registers are denoted by $\mathsf{L}_{\mathsf{T}_p}$, $\mathsf{H}_{\mathsf{T}_p}$, $\mathsf{STK}_{\mathsf{T}_p}$ and $\mathsf{REG}_{\mathsf{T}_p}$ respectively. We refer to the tuple

$$\mathsf{S}_{\mathsf{T}_p} : <\mathsf{R}_{\mathsf{T}_p}, \mathsf{W}_{\mathsf{T}_p}, \mathsf{RW}_{\mathsf{T}_p}, \mathsf{L}_{\mathsf{T}_p}, \mathsf{H}_{\mathsf{T}_p}, \mathsf{STK}_{\mathsf{T}_p}, \mathsf{REG}_{\mathsf{T}_p}>$$

as the *state* or *execution context* of a *live* transaction T just before program point p (the subscripts T_p or p may be dropped when not necessary).

When a checkpoint is restored due to a conflict, it begins execution in exactly the same context as the context of the transaction when this checkpoint was saved. This requires saving a transaction's state at some arbitrary point in its execution and restoring it at some other instant during its lifetime. This is straightforward to achieve in languages with support for first class continuations but challenging for languages without them. Here we present a form of continuations (for the C/C++ languages) that transactions use to save and restore state during a checkpoint operation.

2.3.1 Persistent First-Class Continuations

For a dynamic program point p in a transaction T, we define a persistent *continuation* that encapsulates a transaction's complete state $\mathsf{S}_{\mathsf{T}_p}$ as defined above, *immediately before* p. By persistent we mean that the continuation continues to exist after p and also after the program stack frame at p ceases to be live (for example, if the function containing p returns to its caller). Each of $\mathsf{R}_{\mathsf{T}_p}$, $\mathsf{W}_{\mathsf{T}_p}$ and $\mathsf{RW}_{\mathsf{T}_p}$ can be saved into this continuation - if we assume that each of them are maintained as ordered lists, then their states can be captured simply as the position of the last inserted element in each of them. In addition, this continuation also captures the transaction heap memory allocations and deallocations and like the read/write sets above, these are restored when the continuation is *activated* on a conflict. In addition to the read/write sets, normal transactions also maintain a write-set for local variables since they have to be restored when a transaction aborts and restarts. This local variable write-set is also captured in the continuation. This continuation is also used

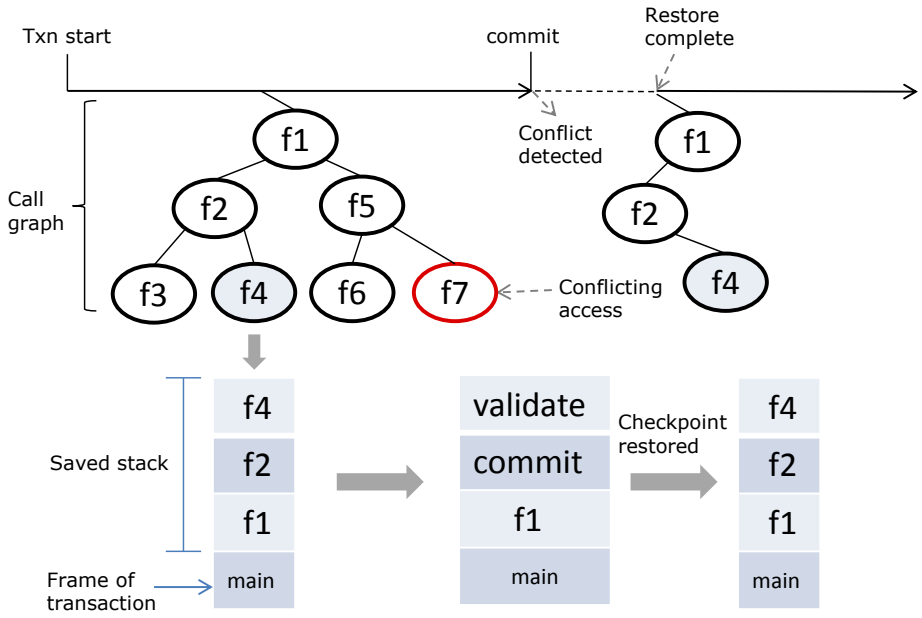


Figure 4: Saving and restoring the state of the stack on a conflict

to record the program stack starting at the frame containing the start of T to the frame at the top of the stack at p. Thus the states of the local variables L_{T_p} - the state of local variables in the current stack frame at p and in all other *live* stack frames underneath, are also recorded. A checkpoint H is said to be registered with its transaction T when execution of T encounters the `CheckpointSave()` call. Then a continuation is created on a transaction-private region of the heap for the checkpoint H that encapsulates S_{T_p}. Figure 4 shows a transaction saving the state of the stack as part of a continuation while in function f4(). Later, while executing f7(), the transaction accesses a transactional variable that at commit time is found to have a conflict. At this point the continuation saved in f4() is restored and execution is resumed from that point.

The compiler pass for inserting checkpoints into a transaction's body is shown in Figure 5(a). This figure also shows the IR output in Figure 6 for the list search function shown in Figure 5(b). The compiler pass processes callers before callees and the call graph is processed in depth-first order. The pass starts with the function

body containing the transaction's boundary (the start and end instructions). The pass begins by inserting a special marker instruction at the beginning of the transaction. This marker instruction essentially stack allocates (`alloca`) a transaction-local *marker* variable seen in lines 5-6 in Figure 6. When a continuation is saved, the state of the program stack $\mathsf{STK}_{\mathsf{T}_p}$ at point p in that transaction is saved relative to the state of the program stack at this marker variable. At runtime, when a checkpoint is saved at dynamic program point p when the stack pointer register contains `esp`, the stack is saved by copying the portion of the stack between `esp` and the `marker` variable into the continuation. So all the stack frames that are live at p are recorded.

The pass then inserts a call to the checkpoint save operation `CheckpointSave()` before each transactional load operation it encounters (line 7 in Figure 5(a) and line 28 in Figure 6). In typical transactions, transactional loads to shared values are the most frequently occurring transactional operations and are also the transactional operations which have the highest likelihood of experiencing a conflict. So it makes sense to insert checkpoint operations before transactional loads. Other systems such as the one in [34] instead insert checkpoints before specific store operations. This makes fine-grained control of checkpointing less feasible (since stores are relatively less frequent than loads) and also means that these system will not directly help the performance of transactions that are either read-only or are read-intensive. On the other hand, while associating checkpoints at specific chosen loads gives us better coverage of the transaction, saving a checkpoint at every dynamic transactional load is obviously prohibitively expensive in practice. In our system, these checkpoint operations inserted before every load are treated only as potential program points to save a checkpoint. At runtime a transaction decides whether a checkpoint is actually saved by evaluating a few simple heuristics. This and other techniques to reduce state saving overheads are described below.

```
1  CheckpointTxnRegion(Function *F, Inst *start,
    Inst *end, Inst *marker) {
3   if(marker == NULL) {
     Inst * marker = InsertMarkerAt(start);
5    txnStackDepth = start->stackdepth();
    }
7
     foreach Transactional Load Inst i∈(start, end)
9     state_opts = i->stackdepth() - txnStackDepth;
      InsertCheckpointBefore(i,marker, state_opts);
11
     foreach Transactional Call Site c∈(start, end)
13   {
      callee = c->targetFunction;
15    if (!ProcessedCallTargets.add(callee)) {
       CheckpointTxnRegion(callee, c->start, c->end,
17     marker);
     }
19 }
   -------------------- (a) -------------------
21 atomic {
    int big_array[100];
23  list_find(key, list);
   }
25
   // list->head is read-only
27 node_t *list_find (int key, node_t *list) {
    node_t *x = list->head;
29  for(;x;) {
     if(x->key == key)
31    break;
     x = tm_read(&(x->next));
33  }
    return x;
35 }
   -------------------- (b) -------------------
```

Figure 5: (a) Overview of compiler pass to checkpoint transactional regions (b) routines for atomic list search

```
{
call @tm_start(..)
 ; marker could have been alloced here
 %big_array = alloca i64*100
 %m = alloca i64
 store %m, %txn->marker,
 call list_find(%txn, %key, %head)
call @tm_end(..)
}
define node_t* @list_find(Thread_* %txn,
 i64 %key, node_t* %head)
{
entry:
 %x = alloca node_t*;
 %1 = load %head
 store %1, %x;
 br label %bb2
bb2:
 %12 = load %x;
 %13 = icmp ne %12,null
 br %13, %bb, %bb3
bb:
 %4 = load %x
 %5 = load %x->key
 %6 = icmp eq %7, %8
 br %6, %bb3, %bb1
bb1:
 %9 = call @CheckpointSave(%txn)
 %7 = call @tm_read(%txn, x->next)
 store %7, %x
 br %bb2
bb3:
 %8 = load %x
 ret %8
}
------------------- (c) --------------------
```

Figure 6: Simplified IR generated by the compiler pass in (a) for the code in (b)

2.3.2 Reducing State Saving Overheads

Saving the local and shared read/write sets, heap alloc/deallocs and registers at a point in a transaction takes a constant amount of space and time and as a result is relatively inexpensive. Saving a potentially unbounded program stack however, is not and the amount of state that is to be saved on a checkpoint save operation can be significant especially if this save is deep in a call chain (as in the case of the checkpoint save operation in function f7() in Figure 4). Moreover transactional loads are quite frequent and since we augment every load with a potential checkpoint save operation, reducing the amount of state saved on each checkpoint and reducing the frequency of checkpointing itself are critical to performance. Our implementation of the compiler pass outlined in Figure 5(a) performs a few state-saving optimizations to this end that are not illustrated in this figure but which merit discussion.

The stack allocation of the marker variable is typically done just before the transaction's start (Figure 5(a) line 4). That is, during a checkpoint save, everything on the program stack from the current stack register to the last allocated stack variable is saved by the checkpoint. In the first optimization the compiler attempts to eliminate saving the regions of the stack that are not written to in the transaction. For example the stack allocation of the array `big_array` in the Figure 6 is not written to in the transaction but may be referenced later in that function. If the marker variable were allocated normally just after the transaction's start, every checkpoint save operation would also save the state of this `big_array`. Instead, the pass attempts to lower the position of marker on the stack such that it is allocated after this array - in line 5 instead of line 3 in Figure 6.

Before the pass inserts a checkpoint in line 7 in Figure 5(a) it checks if that particular access occurs in the same stack frame as the transaction's start and end. If so

then the portion of the stack frame that is to be saved and restored is significantly reduced (modifications to the stack allocated local variables are tracked by the transaction itself and so need not be saved here). Additionally it then checks if any of the local variables in the transaction's enclosing scope can be *written* to in the transaction. If it can be guaranteed that they are not then the contents of the stack need not be saved at all. This optimization is especially beneficial for small transactions that do not access any stack state (such as transactions that atomically increment a shared global counter).

Runtime Heuristics

The compiler pass inserts a checkpoint save operation before every transactional load, *at runtime* these calls to the checkpoint save operation evaluate a set of heuristics to decide if a checkpoint is to be saved before the dynamic load about to be executed.

1. **Age of the transaction:** One heuristic we use is the number of dynamic transactional loads/stores that the transaction has executed so far. This metric is often a good indicator of the amount of work that the transaction has performed so far, since we do not want very short running transactions to execute potentially costly checkpoint save operations. Therefore a transaction will only save state at a checkpoint operation if the number of dynamic loads/stores so far is greater than some threshold n_{ldst}.

2. **Time elapsed since last checkpoint**: The second heuristic controls the frequency of saving checkpoints by checking if the current checkpoint save operation is atleast n_{freq} number of loads since the last one. A value of $n_{freq} = 1$ would mean that a checkpoint save would be performed for every dynamic transactional load or store.

3. **Total number of active checkpoints**: The third heuristic checks if the total

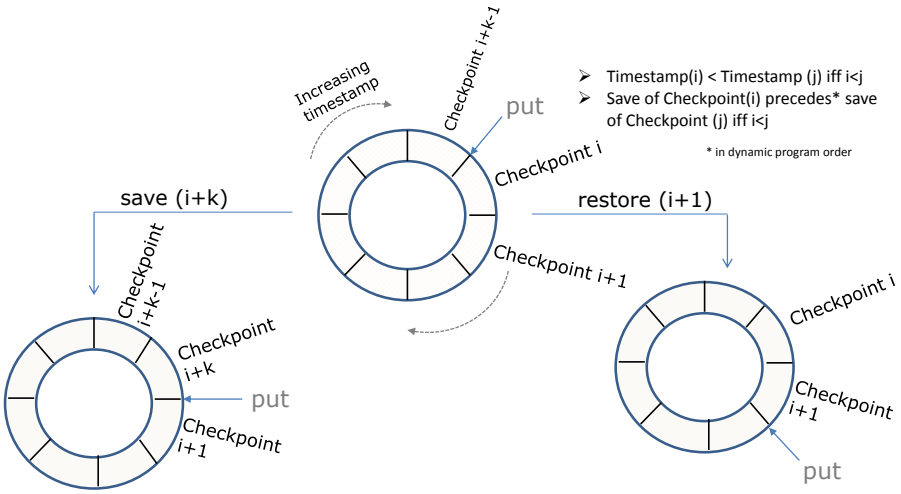


Figure 7: A transaction-private, circular buffer with k entries for saving and retrieving ordered checkpoints

number of *active* saved checkpoints for a transaction is less than some threshold n_{saved}. This is to reduce the cost of picking a checkpoint to restore during a conflict and also to control the memory footprint for transactions that save a large amount of state on each checkpoint.

4. **Average abort rate of the transaction**: In low-contention scenarios where a transaction aborts rarely, the benefit of saving and restoring checkpoints is low. On the other hand, for a transaction that is experiencing a very high abort rate especially after it has completed a significant amount of work, saving and restoring checkpoints can help reduce the amount of work it rolls back. This heuristic compares the number of aborts a transaction has experienced so far to a threshold and decides whether to save a checkpoint at an upcoming load or not.

All four of the thresholds described above are fixed on a per-transaction basis at compile-time in our implementation. However making these thresholds tunable by the transaction itself may be useful in some cases. For example, if a transaction is experiencing a high rate of aborts due to high contention-levels, then it may accelerate

its own rate of checkpointing so as to avoid these aborts.

2.4 Runtime Support

Checkpoint Chaining When a transaction experiences a conflict it attempts to find the *latest* checkpoint that was saved before the access that caused the conflict. To do this, each transaction maintains a private timestamp which is simply a monotonically increasing counter that is incremented every time the transaction makes a transactional load or a store (note that this timestamp is distinct from the transaction's *clock* which is used to validate accesses in STMs that use global clocks). When a checkpoint is saved, this checkpoint is tagged with the transaction's timestamp at that instant and added to an ordered list of saved checkpoints. On a transactional access, the item being added to the transaction's read/write sets is also tagged with the transaction's timestamp at the time of the access. This allows the runtime to efficiently find the latest checkpoint that occurred before a particular conflicting access - it simply iterates over the ordered list of checkpoints and finds the one with the highest recorded timestamp that is also lower than the timestamp than the read/write set element is tagged with. The runtime chooses this checkpoint to restore to since it represents the last known valid state of the transaction as far as this particular access is concerned. The transaction then validates all the read/write set elements that are tagged with a timestamp *lower* than this element and if successful, restores the checkpoint. This validation step is to ensure that when the transaction is restored to this saved checkpoint, its read/write sets at that point are valid and coherent.

One way of storing these timestamps is in a circular-buffer with k-entries as shown in Figure 7. When a transaction saves a new checkpoint, it is inserted into this buffer into the slot pointed to by `put` and `put` is advanced to the next slot (in a predetermined direction, clockwise in this case). So at any instant this buffer holds the totally-ordered last k saved checkpoints. On conflict to an access with timestamp *t'*,

the transaction starts at `put` and iterates in the opposite direction (counter-clockwise in this example) to find a checkpoint with a timestamp $t < t'$. If it finds such a checkpoint, we are guaranteed that there is no other checkpoint with a timestamp t'' such that $t < t'' < t'$. When the checkpoint with timestamp t' is returned, all the other checkpoints with timestamp higher than t' are invalidated since they were saved in a program state that is after t'.

2.4.1 TM Model

The discussion of checkpointing semantics and their execution model so far is independent of the specific TM model. Here we describe the support needed in the TM itself for registering and invoking checkpoints and so we focus on certain types of TM systems for this discussion. At a high-level, the TM model we consider is that of a *lock-based*, *write-back*, software TM that guarantees *opacity*, uses commit-time locking and performs validation at both encounter time (during an access) as well as at commit time. This describes a large variety of systems including TL2 [1], TinySTM [7] and DSTM [9] among others. A thread begins executing a transaction T by calling `tm_start()`. In this step all of T's data structures such as read/write sets, filters etc., are allocated and/or initialized. The global clock is also sampled and the timestamp is stored as T's start time. This clock is simply a monotonically increasing global counter and the start time is used in the conflict detection stage for determining whether a variable accessed during execution of T was concurrently updated by another concurrent thread. The body of T the `tm_read()`, `tm_write()` and related calls for performing speculative accesses to shared data. When finished, T attempts to commit by calling `tm_end()`. This marks the start of the validation (also referred to as conflict detection) phase which we describe in more detail below.

Validation and Restoring Checkpoints: In the first step in T attempts to validate

$\mathsf{R}_{\mathsf{T}_p}$ and validate *and* acquire a lock on each element in its $\mathsf{RW}_{\mathsf{T}_p}$ and $\mathsf{W}_{\mathsf{T}_p}$ sets. The outline of this step for $\mathsf{RW}_{\mathsf{T}_p}$ is shown in Algorithm 1. For each element e in $\mathsf{RW}_{\mathsf{T}_p}$ its current version number is compared to T's start time. If the former is greater, then e was updated by another transaction i.e., e is invalid and T is aborted. If not, it checks whether e is currently locked by another concurrent transaction. If it is then the latter will most likely commit sometime in the future and update e thereby rendering T's copy invalid. Thus in this case too it aborts immediately. If e was both valid and not locked then T attempts to acquire a lock on it and aborts if it is not able to. This process is repeated for every element in its read-write and write sets. In the next step the read set for T is validated. This is similar to the above except no locks are acquired - for an element in the read set, if it is not currently locked and its version number is lower than T's start time then the element is considered valid. If all the elements of the read/write sets have been found to be valid and all the locks are successfully acquired, then T is considered to have been validated and it moves into the write-back stage. In this stage, the values computed by T and produced into its local write buffer are finally committed to main memory. After this, the transaction has finished committing and releases all the locks it acquired in the validation step above.

A checkpoint is invoked when the validation of its parent transaction encounters a conflict. A high-level outline of the *commit-time* conflict detection stage for variables that are *read-and-written* is shown in Algorithm 1. Lines 6 - 16 are related to the corrective conflict resolution while the rest of the algorithm describes the standard detection and resolution scheme in our lock based optimistic concurrency control system. The outer for-loop (which is also part of normal conflict detection) iterates over the elements in read/write set and validates and locks them. If validation (`isValid()`) and lock acquisition (`getLock()`) for a particular element are both successful, that element is marked as *valid* (`markValidated()` in line 4). If either of these steps

fails for an element then the transaction attempts to find the latest checkpoint that was saved after that particular access (`chooseCheckpoint()` in line 6). If no such checkpoint can be found, then the transaction aborts. Otherwise, it validates the portion of its read-set upto the conflicting element (`validateReadSetUntil()`) in line 7). This prevents the transaction from restoring to a state that is invalid (specifically, a state in which its read-set has been invalidated). It then drops all the locks it has acquired so far (`DropLocks()` in line 9), samples the global clock and finally restores the checkpoint that was found (line 11). After restoring a checkpoint the transaction may modify its newly restored read/write sets in two ways. It may extend the read/write sets by calling `tm_read()` or `tm_write()`. That is, new elements are created and added to the respective tails of its read/write sets. Therefore these new elements are in turn validated as the outer for-loop in Algorithm 1 reaches them when the transaction attempts to commit again. Secondly the transaction may modify the values cached in the elements in read-write or write-only sets by writing to memory locations it wrote to before the checkpoint restore. This does not affect whether an element is or will be successfully validated. It also does not invalidate an already validated element since the transaction would have acquired a lock for that element before it began executing. Validating the transaction and invoking checkpoints for conflicts to read-only and write-only (or write-and-read) elements proceeds in a similar way except no locks are acquired for memory locations that are read-only. Similarly the *encounter-time* validation algorithm for read/write transactional accesses is similar to the one above except that no locks are acquired.

Multiple Conflicts During a transaction's execution or validation, multiple locations that it has accessed may have been invalidated. In practice this is quite common and the validation/restoration scheme presented here handles this case seamlessly. Even though multiple read/write set elements have been invalidated a transaction only detects conflicts one at a time. When a conflict for a particular access has been

Algorithm 1 Conflict Detection for RW set

```
1: // To validate locations that are read and then written:
2: for all e ∈ T→RWSET do
3:   if isValid(e) && getLock(e) then
4:     markValidated(e)
5:     continue
6:   else if (c=T→ChooseCheckpoint(T,e)) then
7:     if ValidateReadSetUntil(T,e) &&
       T→retries<MAX_RESTORES then
8:       T→retries++
9:       DropLocks(T)
10:      readGClock(T, e)
11:      RestoreHandler(T, c)
12:    else
13:      return TABORT
14:    end if
15:  else
16:    return TABORT
17:  end if
18: end for
19: T→HoldsLocks = true
```

detected, before the appropriate checkpoint is restored the transaction attempts to validate its read/write set as it existed *when the checkpoint was saved.* If there were (not yet detected) conflicts to locations accessed before this particular access, then this validation step will fail and the transaction simply aborts. In the second case, if there were (not yet detected) conflicts to locations accessed after this particular access, then these conflicts can be safely ignored since the checkpoint restore would restore the transaction to an instant when accesses to these locations did not yet occur. After the checkpoint is restored, these same locations may be once again accessed and they will be validated as they would be in a normal transaction.

2.5 Safety

A TL2-like TM has the following properties:

1. Memory locations are added to the R, W, RW sets in the order in which they were *first accessed*. For elements in each of these sets we define an order $e_j \prec$

e_i if e_j appears before e_i in the set.

2. A transaction never reads inconsistent state.

3. Transactional reads or writes to the same memory location are *not* collapsed.

Informally, T can commit successfully if the following sequence of checks are successful
i) R is coherent and
ii) RW & W are coherent and locks can be acquired on all their elements and
iii) R is still coherent

Consider step (ii) during commit-time validation for T. According to the algorithm above, T aborts if lock acquisition failed for some word $e_i \in$ RW or if the version number changed since it was read i.e., it is no longer coherent. Consider the latter case. When this conflict is detected,

$$start_T < version_{e_i} \text{ and } version_{e_i} \leq globalclock \qquad (0)$$

where $start_T$ is T's start time, $version_{e_i}$ is the version of e_i last written and *globalclock* is the current value of the global clock. Since the conflict detection validates elements in order, this means

$$\forall e_j \prec e_i \in \mathsf{RW}\text{: } e_j \textit{ is valid} \qquad (1)$$

Before a checkpoint is restored the R is validated until e_i. Therefore

$$\forall e_k \prec e \in \mathsf{R}\text{: } e_k \textit{ is valid} \qquad (2)$$

After the checkpoint is restored, the last elements in RW and R in these newly restored sets are the ones immediately before e_i in those sets before the checkpoint was restored. And therefore from (1) and (2), the newly restored R and RW sets are *coherent* and *valid* and therefore the transaction T is in a *consistent* and *valid* state.

Moreover, since its read/write sets are valid at that point, the transaction can safely read the global clock and move its own $start_T$ forward to $start'_T$ where

$$start'_T \geq globalclock \qquad (3)$$

Restoring the transaction can eliminate the conflict on e_i as follows. After the transaction restore, lets say the transaction accesses the memory location corresponding to e_i again. From (3) and the second part of (0),

$$version_{e_i} \leq start'_T \qquad (4)$$

So this new access to the memory location corresponding to e_i is guaranteed to see a valid version of e_i and this access is guaranteed to not result in an encounter-time conflict.

After a checkpoint restore for e_i, the transaction may have performed speculative loads or stores on new memory locations. These new accesses are simply appended to the list of yet-to-be-validated accesses (just as would happen in a normal speculative access in T) and are locked and validated much like e_i - when the transaction ultimately attempts to commit, each of the read, read-write and write-sets are re-validated in their entirety. In our TM model (which corresponds to a TL2 like STM), transactional writes to private (local) heaps locations are logged in a manner similar to transactional writes to shared heap locations. That is, the transaction maintains a separate "local write" buffer that logs the values being written. These values written are committed in order when the transaction commits successfully. So the entire series of values being written to a transaction-private memory location are logged and therefore a checkpoint restore can restore these values to any point in the transaction's execution. The checkpoint and restore mechanisms handle these local read/write sets the same way they handle $\mathsf{R}_{\mathsf{T}_p}$, $\mathsf{RW}_{\mathsf{T}_p}$ and $\mathsf{W}_{\mathsf{T}_p}$. However unlike the read/write sets, the transaction-local heap accesses need not be validated and no locks need be acquired on them.

2.5.1 Opacity

When specified inside transactions that satisfy the *Opacity* property [31], checkpoint operations also satisfy this property. Informally this means:

- **Atomicity**: All operations performed within a *committed* transaction before and after all checkpoint restores appear as if they happened at some indivisible point during an instant between the start of the transaction and its commit.
- **Aborted State**: The effects of an operation performed inside an *aborted* transaction before or after a checkpoint operation are never visible to any other transaction.
- **Consistency**: A transaction always observes a *consistent* state of the system, before and after all checkpoint restores.

2.5.2 Isolation:

A transaction before or after a checkpoint restore only observes *consistent* state, i.e., it is guaranteed to not see any updates that have not been committed by a *live* concurrent transaction. Also, inserting checkpoint operations into a transaction at compile-time does not require knowledge of either (a) other concurrently executing transactions or (b) how the other transactions may have modified variables that caused the conflict (which invoked this checkpoint) or (c) how many other transactions committed between the start of this transaction and the invocation of the checkpoint. However, even though checkpoint handlers are *semantically transparent*, using them results in a different global ordering of transactions than when they are not used and also permits a different subset of all conflict-serializable schedules.

2.6 Experimental Evaluation

We implemented the compiler pass in for generating checkpoint operations and optimizing them in the LLVM [15] compiler (v2.4) and the runtime support for checkpoints in the TL2 TM system [1]. In this section we analyze the performance impact of applying these corrective checkpoint restores through experiments on parallel transactional workloads in the STAMP suite [3]. The `list` program is a library component of STAMP that is used extensively in many workloads in the suite. The `counter` program implements a simple shared counter updated concurrently by several threads, a commonly occurring parallel programming artifact. We used an unmodified TL2 STM [3] as our baseline optimistic concurrency control system. Both the unmodified TL2 baseline and our checkpointing TL2 STMs use write-buffering, lazy-validation and commit-time locking. All workloads were compiled using LLVM and gcc-4.3.3 for final code generation, with the default optimization flags for each workload. We ran all experiments in Linux on a machine with dual Intel Xeon X5500 4-core processors in which each was core clocked at 2.93GHz and each core also had hyperthreading enabled (for a total of 16 contexts). To reduce interference due to scheduling each thread was bound to a specific processor core uniformly. All the workloads were executed with the standard reference inputs if defined (else the inputs are described in the discussion below). The baseline versions of the programs use normal optimistic concurrency control in transactions using an *unmodified* TL2 STM and hence do not save checkpoints or restore them on conflicts. All timing measurements were the average of 5 runs. The plots in Figure 9 show the speedups obtained using checkpoints - we use the metric *speedup* to refer to the **ratio of the execution time for the baseline case (with unmodified TL2) to that of the execution time using our compiler and runtime scheme for the same number of threads**. We experimented with several values for the set of parameters (n_{ldst} ,n_{freq}, n_{saved}, n_aborts) for the heuristics for reducing state saving overheads but due to space limitations we

report results for the set of values (1,256,32,1) except for the `counter` program for which we used (1,1,32,1).

counter The `counter` program implements a simple shared counter that is incremented by concurrent threads. This is a commonly occurring parallel programming construct in many parallel programs. The program has a single transaction that simply performs a read, increment and write to the counter. The checkpoint save for this transaction does not have to save any stack state. When the transaction validates its read-and-write access, it acquires a lock on the address of the counter and after a restore, it simply executes the entire transaction body while retaining the lock and then validates successfully and commits. This corrective action reduces the abort rate quite significantly as is seen in Figure 11. The execution time speedup due to this ranges from 1.4X to over 4X. Although the amount of work done in each transaction is small, the amount of contention for this program is very high. We noticed that for 16 threads even though the number of aborts are reduced (meaning many of the checkpoint restores are successful), the overhead of executing them outweighs the benefits for this level of contention. There is very little state saved on a checkpoint as shown by the data in Table I. Moreover, almost every conflict can trigger a restore in this program leading to a high number of average checkpoint restores per successful commit as shown in Figure 10.

list The `list` program implements a single linked list without duplicate key values. This program (or rather the linked list library used by this program) is used extensively in the other STAMP benchmarks. The program creates and initializes an initial list and launches several threads which perform concurrent operations on this list. An operation can be one of `insert`, `find` or `remove` with a specified key to insert, find or remove with each of them corresponding to 20%, 60% and 20% respectively of the total number of operations performed on the list. Each of these

operations is implemented as a transaction. Given a key `k` to insert the `insert` routine iterates over the list and finds the right position to insert this key into. Then the actual modification of "`next`" pointers takes place as in standard list insertion. Similarly the `remove` routine iterates over the list to find the element to remove. The insert and remove routines also increment and decrement the `size` of the list. Since all three operations involve traversing through the list, most of the time spent in transactions in this program is spent in iterating through a list looking for a key (similar to the code shown in Figure 5(b)). As each new element is encountered during iteration, an optimistic load is performed on its `next` field. If there is a conflict on this field then after the checkpoint is restored the new next pointer is loaded (using `tm_read`) and the search is resumed. The reduction in aborts due to this corrective action is significant as seen in Figure 8. The improvement in execution time ranges from 1.4X to 3.2X (Figure 9). The speedup is limited by the overheads of validating state before restoring a checkpoint - during the corrective action many newly committed pointers may be encountered which will be added to the read/write sets and which will have to be validated. Moreover, if a conflict occurs on reads to these newly committed pointers a checkpoint may be restored again. Therefore there may be several checkpoint restores for each successful commit. This is supported by the high number of restores per successful commit shown in Figure 10.

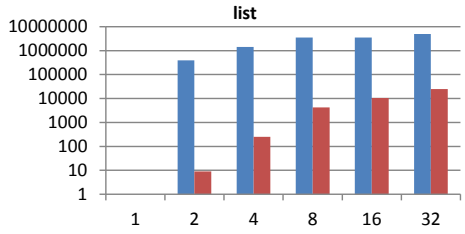


Figure 8: Aborts Vs. Threads in `list`

kmeans The `kmeans` program implements a transactional version of the popular Kmeans algorithm using optimistic concurrency control [3]. This workload contains

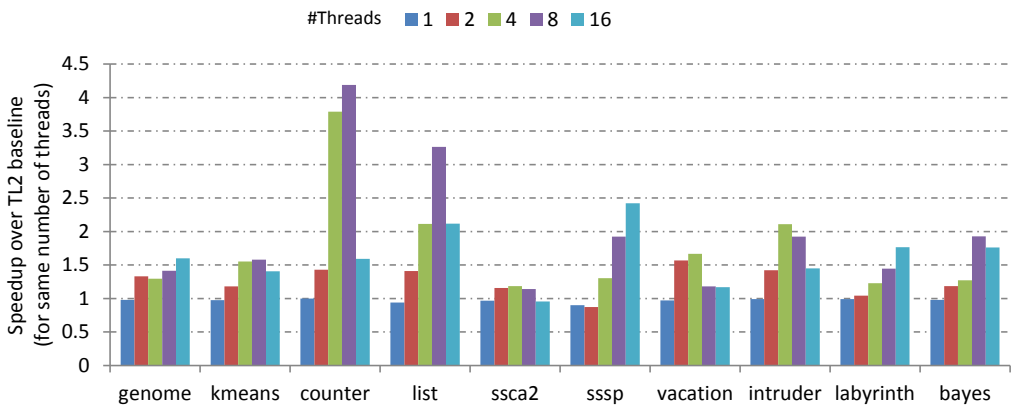


Figure 9: Speedup in execution time over a parallel TL2 baseline version of the program running with the same number of threads (each bar shows the ratio b_n/c_n where b_n is the wall clock execution time of the plain TL2 version of the program and c_n is the execution time of the checkpointed version).

a total of three critical sections implemented inside transactions. The first two add a value to a shared scalar variable. The checkpoint operations for these transactions are similar to the one discussed in the example of incrementing a shared counter. Most of the time spent in transactions is in a third transaction in the `work()` function. This transaction begins inside an outer loop and contains a loop within itself which updates elements in an array of numbers. Most of the conflicts suffered by a transaction are due to accesses to shared values inside this inner loop. The average cost of each conflict is high too - a conflict on an access inside this loop means that the updates made to the array so far are discarded and the transaction restarts updating the array from scratch. With a checkpoint the transaction instead restores state to the point just before the conflict therefore reducing wasted work. Additionally, since the transactional accesses are in the same stack frame as the transaction's start, very little state is actually saved (Table 2.6) since checkpointing the read/write sets takes a constant amount of time and space, irrespective of their size.

The reduction in abort rate for `kmeans` is shown in Figure 11. Note that the Y-axis in the figure uses a log-scale. The abort rate is reduced by several orders of magnitude in some cases when using checkpoints. Figure 9 also shows that there is

Table 1: All numbers are for 4 threads. Column (A) is the **percentage of checkpoint restores** that ultimately resulted in a commit of a transaction that would have otherwise aborted. Column (B) is the **average size** in bytes of the state saved by a checkpoint operation. Column (C) is the **average call stack depth** of a checkpoint save operation, relative to the transaction's own stack frame

Program	**Column (A)**	**Column (B)**	**Column (C)**
counter	84.46	8	0
list	71.48	78	1
kmeans	82.18	16	0
ssca2	4.77	16	0
genome	66.18	64	2
sssp	8.02	92	3
vacation	73.23	64	3
intruder	1.61	178	6
labyrinth	54.05	112	2
bayes	13.8	198	2

also a significant reduction in running time - up to 1.58X in the case of 8 threads.

ssca2 Most of the critical sections in `ssca2` are small and perform simple operations such as increments or adding scalar values to shared variables. However most of the time spent in this program is spent in one particular critical section which is inside a 2-deep loop nest. Corrective handling of conflicts can therefore be very beneficial here. However the transaction also contains control flow that is predicated upon shared variables. The future transactional accesses that will be performed depend strongly on the results of the accesses already performed. Hence for this transaction, when the checkpoint is invoked it rebuilds most of the transaction's state (according to the new control flow paths). Moreover, although the size of the state saved is quite small as seen in Table I, the short sizes of the transactions and their high frequency means that checkpointing and restoring them results in high overheads (the overhead of performing a checkpoint save is high relative to the amount of work done in each transaction instance). This is reflected in the experimental results we obtained which are shown in Figure 11 - the number of aborts is reduced significantly but as can be

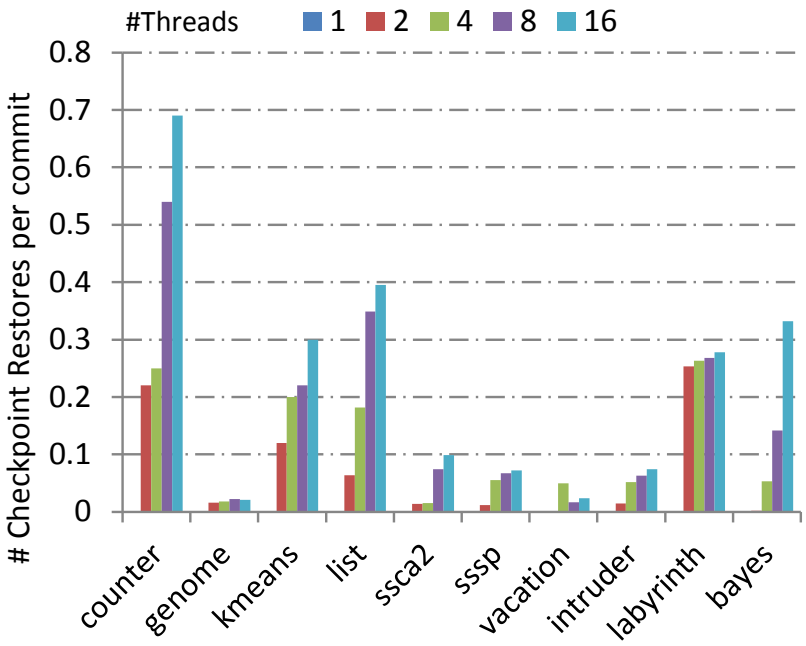


Figure 10: Average number of checkpoint restores successful commit

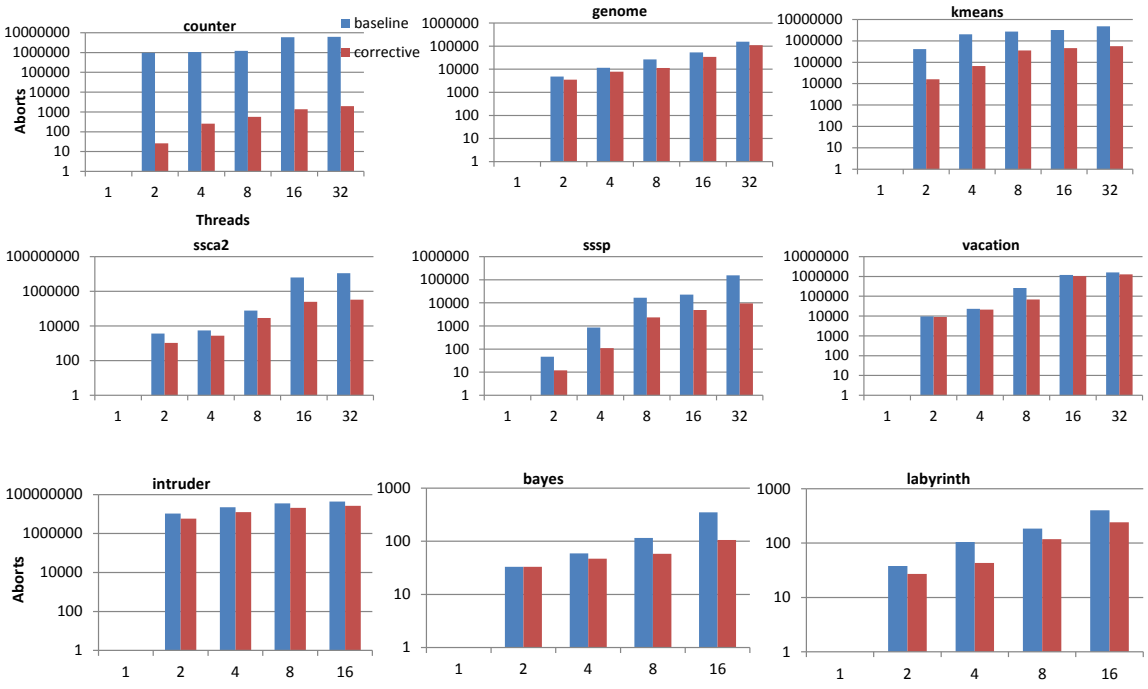


Figure 11: Aborts

seen in Figure 9 the maximum speedup in execution time is about 1.18X.

genome The `genome` benchmark implements a gene sequencing program that reconstructs the gene sequence from segments of a larger gene. There are several transactions for which checkpoints are generated - two of them together account for a significant fraction of the total time spent in transactions. These transactions perform operations on a shared table data structure which is in turn backed by a concurrent linked list. Therefore the checkpoints for these transactions are similar to the checkpoints for the optimistic concurrent list operations discussed in the `list` program above. The speedup for this program due to this corrective conflict resolution ranges from 1.14X to 1.59X.

SSSP The `SSSP` workload consists of a parallel transactional implementation of Dijkstra's shortest path algorithm. The program consists of multiple threads which execute a number of steps in each of which they perform several updates and queries on a dense graph. A query operation specifies a vertex for which the shortest path from the source is returned while an update changes the length of an edge. The graph being manipulated contains 300 vertices and is densely connected. Each query involves an $O(n^2)$ amount of computation and a checkpoint is quite effective in amortizing this cost over updates. The level of connectivity in the graph plays a significant role in the amount of state that has to be saved and restored in the checkpoint. The speedups for this program range from about 0.87X to 2.42X. For sparse graphs we expect the performance improvements to be higher since a change to an edge weight will result in fewer number of successor vertices being examined.

bayes The `bayes` program implements an algorithm for learning Bayesian networks from observed data. The speedups for bayes were significant - almost 2X with 4 threads. This program contains several transactions of varying sizes ranging from

short transactions incrementing counters to long running transactions that query shared lists. Most of the contention and aborts came from a long transaction in the `TMfindBestInsertTask()` function which is read-only and iterates several shared linked lists while other transactions are modifying them. As in the case of the `list` program, checkpoints are effective in avoiding wasted work in this program by restoring the state of instances of this read-only transaction to an earlier point in their execution rather than aborting and starting from scratch.

Vacation The `vacation` program from STAMP implements a travel reservation system powered by a non-distributed database. The database consists of several tables which are implemented as Red-Black trees. The reduction in aborts was not as dramatic for most configurations as shown in Figure 11. The best speedup over the baseline version was noted for the case of two threads - approximately 1.54X shown in the plot in Figure 9. Checkpoints are most effective in improving execution time for programs in which the save point occurs after some significant work has been done in the transaction (work which would be discarded in the case of an abort but which is salvaged if a checkpoint is restored instead). In this program the highly contended accesses occur fairly early on in the transactions and therefore less work is discarded due to aborts.

Intruder The `intruder` program implements a signature based network intrusion detection system. The targets of contention for this program are several queue, list and tree datastructures that are used in the network packet capture, reassembly and detection phases. In [96] the authors observed that a "push" operation onto a shared queue operation was the main source of contention in this program. Additionally this push operation occurs towards the end of a long running transaction which means that quite a bit of work is wasted due to conflicts on this operation. With checkpoints the transaction simply restores its state to an earlier point in its execution therefore

Table 2: Reduction in number of memory references due to checkpointing. All numbers are for 8 threads.

Program	**# Memory References (Baseline)**	**# Memory References (Checkpointed)**	% Reduction
list	20314520865	22679762226	-11.6
intruder	45977932488	45654459051	0.70
kmeans	9365082514	6858431392	26.76
ssca2	12094949226	15563038395	-28.67
genome	4452177612	4208840881	5.46
vacation	17152213238	21261187025	-23.95
labyrinth	21696383510	20939193375	3.48

significantly reducing wasted work. However the average stack depth of a checkpoint save is 7 (Table I) meaning the amount of state saved is also high. Inspite of this checkpoints improved execution time significantly - nearly twice as fast as baseline TL2 with 4 threads.

labyrinth The `labyrinth` in STAMP implements Lee's algorithm for finding the shortest distance between two given points on a grid. All the transactions contain a small high-contention critical region that checks the status of a shared flag. If this flag is set, the transaction forces itself to restart without even attempting to commit. Therefore saving a checkpoint for this access would not be useful since when the transaction attempts to commit it has typically already validated itself. However there are other accesses that are served well by checkpointing and this program shows moderate speedups of upto 1.75X (or about 42%) over the TL2 baseline.

2.6.1 Note on overheads

The magnitude of performance improvement from using checkpoints depends on the cumulative cost of state saving and restoration, relative to the cost (including wasted work) of a complete abort. We found that the cumulative amount of state saved

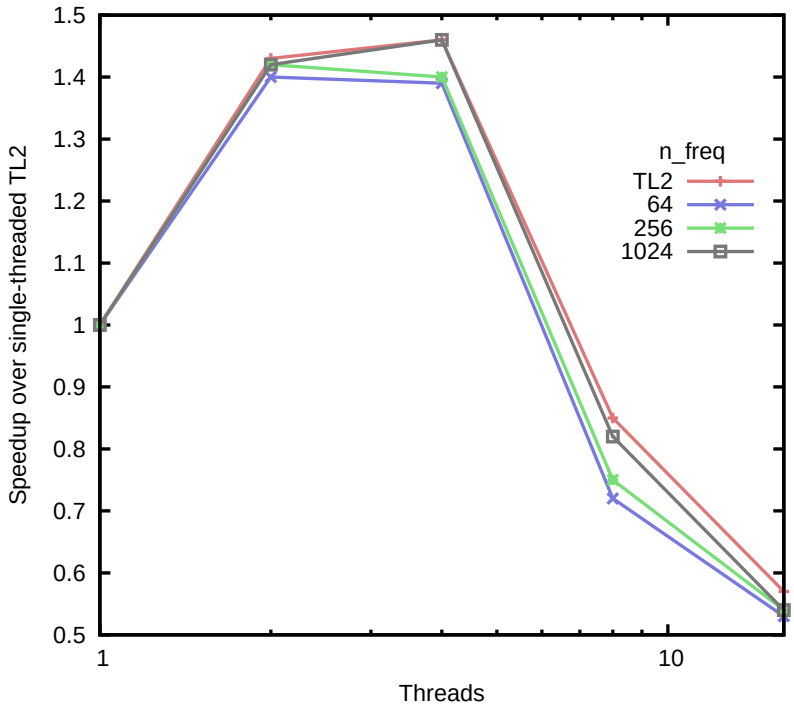


Figure 12: Overhead of checkpoint saving in an execution of `list` with very high-contention - 60%/20%/20% `find/insert/remove` and a small key range. Each of the lines shows speedup over single-threaded TL2 for a specific value of *n_freq*, the frequency of checkpointing as described in Section 3.2

strongly correlated to the speedups. While transaction internal state such as read-/write sets and speculative heap alloc/deallocs were quite efficient to checkpoint, the cost of saving stack frames was especially influential on performance. Therefore transactions with accesses occurring in the same stack frame without any local variables being modified, performed best. Additionally our technique is better suited to long running transactions that would lose a substantial amount of work on an abort. The frequency of saving checkpoints has an interesting influence on running time. If this frequency is too high, the state saving overheads dominate and performance can be poor. However if this frequency is too low, a checkpoint restore may restore state to a point very early in the transaction therefore minimizing the reduction in wasted work. This suggests that there may be a program specific (and input data set specific) sweet spot for this frequency - a question that we intend to explore in future work.

The plot in Figure 12 shows the overheads of saving checkpoints for a high-contention list that is used with a very small key range. The overheads are all quite small with the higher frequency of saving checkpoints resulting in slightly higher overheads (this plot does not include the overhead of finding and restoring a checkpoint, only that of saving one). The small amount of state to be saved per checkpoint is the principle factor in these low overheads. The Figure 9 shows that for all the programs the overhead of saving checkpoints in a single-threaded execution is not significant. This is because of the "contention" heuristic described in Section 3.2. This heuristic throttles the rate of checkpoint saving when the average abort ratio is low. Since in a single-threaded case the abort ratio is zero, effectively no checkpoints are saved.

2.7 Conclusions

In this chapter we presented a compiler-driven conflict recovery scheme using which a transaction that has been invalidated due to one or more conflicts can attempt to recover from them with the help of checkpoints that restore the transaction's state

to a previous intermediate point in its execution and execute from that point. We described compiler optimizations to reduce the amount of state saved by these checkpoints and runtime support for finding and restoring a checkpoint. Our experimental evaluation shows that using such checkpoints reduced the number of aborts by several orders of magnitude for some programs and speedups of up to 4X in execution time on a **real machine**, relative to transactional programs that did not use them. One interesting avenue for future work is a cost model of transaction execution that can be used at runtime to decide whether a particular program location is cost-effective for saving a checkpoint - a host of factors from the depth of the call stack at that point, to the amount of work done so far in the transaction, need to be evaluated to guarantee that a save/restore will benefit performance. Compiler analyses especially points-to analyses can be very useful in reducing the amount of state (especially, thread-local stack state) that is saved and restored.

CHAPTER III

IRREVOCABLE TRANSACTIONS VIA STATIC LOCK ASSIGNMENT

Generally in systems that provide *pessimistic* concurrency control, critical sections attempt to acquire locks on all shared data they access, before they begin. Thus when they begin executing, they are guaranteed to be *conflict-free* due to the mutual exclusion provided by the locks they acquired. These systems are *pessimistic* in the sense that they try to preempt conflicts from even occurring by acquiring locks on a conservatively estimated set of shared memory locations (note that the notion of pessimism is distinct from the notion of eager-locking or encounter-time locking as employed in many TM systems).

In contrast, in optimistic TM systems, each transaction begins and continues to execute speculatively until it experiences a sharing conflict with another concurrent transaction. When such a conflict occurs, this transaction is aborted - the state it has computed so far is discarded and all side-effects it has produced are rolled-back and the transaction restarts from the beginning.

Providing optimistic execution entails a significant cost since each transaction must now be able to detect a conflict and must also be able to undo its changes and restore its state to when it started. Concretely, this means that each transaction must maintain a set of shared locations it has read and written (the Read, Write and Read-and-Write sets), it must buffer all its writes so that they can be committed only when the transaction has finished executing and has not experienced any conflicts. In addition, each transaction pays the cost of validation - the process of checking whether locations in its Read, Write and Read-and-Write sets have been written to

by other concurrent transactions.

Critical sections in pessimistic-locking systems on the other hand do not pay these costs since they are guaranteed to be conflict free. On the other hand, critical sections that employ pessimistic-locking schemes often suffer from excessive serialization which results from the locks being coarse-grained. That is, the critical section makes a conservative estimate of the shared data items it is going to access once it starts, and acquires locks on them. This stems from the fact that in general the exact set of memory locations that will be accessed is not known at compile-time or even when the critical section starts. So for example a critical section inserting a node into an ordered linked list may acquire a lock on the entire list since the set of nodes that will be accessed is not known in advance.

There are three main high-level factors limiting performance in optimistic concurrency control systems:

1. Load/Store Tracking: Each transaction needs to record several pieces of information for each loads and store. Specifically for each dynamic load or store operation, a typical transaction in a TL2-like STM system records the address accessed, the actual value read from or written to the memory location and the version number of the value. Thus each load/store operation to a shared memory location triggers multiple additional loads & stores (to memory regions that are private to the transaction).

2. Validation: A transaction is required to maintain a coherent view of its read-/write sets and to abort when it discovers it has been invalidated. This involves validating values that the transaction is about to access during a read operation and in many TM systems, also validating the values that the transaction has previously accessed (this validation involves comparing the *version* of the value that the transaction recorded at the time of the access to it current value).

Therefore each read/write operation triggers a validation of the entire set of values the transaction has accessed. For transactions with large read/write sets with thousands of elements, this validation imposes a significant runtime cost.

3. Cost of Aborts: When a transaction discovers it has been invalidated (for example because a concurrent transaction wrote to a memory location that this transaction previously read from), it is required to abort thereby discarding all the computation it has performed so far and restart from scratch. For large transactions in environments with high-contention between threads, the cost of performing computation that is ultimately aborted is quite high. The corrective conflict recovery techniques presented in Chapter II are targetted towards reducing this cost.

3.1 Hybrid Optimistic-Pessimistic Concurrency

Our system is a hybrid of the purely pessimistic and optimistic approaches. In addition to regular optimistic transactions, we allow transactions to execute in *irrevocable* mode, that is, they are guaranteed to not experience a conflict or to abort. When a transaction executes in this mode, we refer to it as an irrevocable transaction as opposed to a normal transaction which we refer to as a revocable transaction. Thus irrevocable transactions correspond to the pessimictic critical regions and the revocable transactions correspond to the optimistic critical sections.

3.1.1 Why irrevocability is important for performance

Apart from providing a safe mechanism for using irrevocable operations such as I/O, irrevocability is relevant for parallel performance as well.

- Irrevocable transactions typically can eschew the overheads of maintaining state such as read/write sets, validating these sets, and of course since they are never rolled back, do not waste work like revocable transactions.

- In many programs a few long-running transactions do the majority of the work in the program. Making these transactions irrevocable may improve overall transactional throughput if the performance of these few transactions has a significant impact on overall program performance.

- Some programs contain transactions which execute relatively infrequently but when they do, they conflict with all the other concurrently executing transactions. For example, a transaction that resizes or rebalances a data-structure such as an RB-tree has a high likelihood of conflicting with other transactions that are performing lookups or modifications on this data structure. Making the rebalancing transaction irrevocable and would avoid potentially multiple roll-backs and would improve overall performance.

An important limitation in the irrevocability support provided by most TM systems is that they allow for *at most* one irrevocable transaction executing at a time. This requirement is necessary since if two irrevocable transactions were allowed to execute concurrently, they may both cause a conflict in the other and since neither can be rolled-back, a fatal fault occurs (in the case of a shared resource being held by one irrevocable transaction and requested by the other and vice-versa, a deadlock occurs which cannot be resolved since both transactions are irrevocable). When irrevocability is triggered for correctness reasons such as a transaction encountering an unrecoverable operation (such as I/O) this limitation is acceptable - in [131] the authors propose compile-time analysis that detects such unrecoverable actions and schedules the transactions containing them such that atmost one transaction with an unrecoverable action is allowed to execute at a time. When we consider the notion of irrevocable transactions for performance however, this limitation is less desirable. As we will show, the performance benefits from promoting transactions to be irrevocable is significant and in many cases it is desirable to make several concurrent transaction

irrevocable.

The compiler-support and TM runtime system that we describe in this paper allows several concurrent irrevocable transactions to run together using a static-lock assignment scheme that allows these transactions to interact safely with each other as well as with other concurrent revocable transactions.

Previous approaches to the problem of inferring locks from atomic sections have relied on points-to or alias analysis. Using these techniques presents a few non-trivial challenges, the most significant being that traditional alias analysis methods require the pointers being considered be in some overlapping scope. Therefore this class to techniques may not always be sufficient for determining interference between accesses to pointers in concurrent threads since they may not share any common scope. In our compilation scheme discussed below, we use the notion of *accessible heap-data structures* that does not require common scope and is interprocedural.

3.2 Design

In this section we outline to steps to inferring lock-sets for transactions in a given program. We start by first identifying the critical sections implemented as transactions. We then compute the set of static data structures that these critical sections can access. We then use these sets in building a *transaction interference graph* that explicitly represents static access conflicts assuming that any two static transactions can be concurrent at run-time. From this interference graph, we identify a set of irrevocable transactions and revocable transactions and finally we determine a lock assignment and locking discipline to synchronize accesses by these transactions at run-time. We assume that the program is *correct* - that is, it is *race-free* and *deadlock-free*. If there are data-races in the program, we assume that they do not affect program correctness or semantics.

3.2.1 *Must* and *May Access* Analysis using DSA

For each static transaction T in the program we identify the set of pointers P_{ref} and P_{mod} that are read and written (respectively) transactionally. That is for each $p \in P_{ref}$, p appears in T as an argument to a transactional load operation and similarly for each $q \in P_{mod}$, q is used as an argument to a transactional store operation in T. Then for each p in P_{ref} and P_{mod}, we find the set of data structures that p can reference. We do this using an interprocedural context-sensitive and field-sensitive *Data Structure Analysis* (DSA) [15]. DSA is a powerful compiler analysis that can identify disjoint instances of data structures and their connectivity properties. It is fully context-sensitive meaning it can distinguish between heap data structure instances created via distinct program call-paths. It also uses an explicit heap model to disambiguate disjoint instances of data structures without succumbing to the drawback of the compile-time heap representation growing very large.

3.2.1.1 Bottom-Up Data Structure Analysis

The DSA step computes a *Data Structure Graph DSG(f)* for each distinct function f in the program summarizing the set of heap data structures accessible from that function. Each node in *DSG(f)* represents a set of dynamic memory objects and distinct nodes in the graph represent disjoint sets of memory objects. We then specialize the graph *DSG(f)* to transactions appearing in the function body of *f*. That is for each transaction T appearing in f, we build a transaction-specific data structure graph *DSG(f, T)* that represents only the set of heap data-structures accessible within transaction T.

Once all the transaction-specific data-structure graphs have been constructed, we compute the set of data structure nodes in $DSG(f, T)$ that can be read or written in the transaction T. For every transactional load or store in T that takes a pointer argument p (corresponding to the memory address to load from or store to), we

construct a points-to set $MustAlias(p)$ that consists of all the data structure nodes in *DSG(f, T)* that p *must* reference. Similarly we also construct another points-to set $MayAlias(p)$ that consists of all the data structure nodes in *DSG(f, T)* that p *may* reference. Each element in these sets is of the form <*Node, accesstype*> where *accesstype* is either R, W or RW. We use the *accesstype* labels to establish the notion of *interference* precisely with multiple readers and writer transactions.

From the $MustAlias(p)$ and $MayAlias(p)$ sets, we construct two sets $T_{MustAccess}$ and $T_{MayAccess}$ that contain the nodes in *DSG(f, T)* that represent the data-structures that can be accessed in the transaction T.

$$T_{MustAccess} = \bigcup_{p} MustAlias(p) where\ p \in P_{mod}\ or\ P_{ref}$$

$$T_{MayAccess} = \bigcup_{p} MayAlias(p)\ where\ p \in P_{mod}\ or\ P_{ref}$$

If $T_{MayAccess} = \emptyset$, then all the data structures that are accessed in T are known at compile-time - the heap regions corresponding to the data structure nodes in $MustAlias(p)$ for each p in T. Therefore a dynamic instance of transaction T can acquire locks on these data structures when it starts and release them when it commits. Therefore all dynamic instances of this transaction can be executed such that they are irrevocable. If $MayAlias(p)$ is non-empty however then we do not know precisely which data structures are touched in T when it executes. After constructing these sets, we use them in constructing a representation of *interference* between transactions and assigning locks to them.

3.3 Transaction Interference Graph

The transaction interference graph $INT = \{V, E\}$ captures the interferences between static transactions due to potential concurrent accesses to shared data structures. Each node in INT is a static transaction in the program - all dynamic instances of a static transaction are captured in a single node in this graph. Two nodes m and $n \in$

V are connected by an edge e if and only if:

$$(\bigcup m_{MayAccess} \bigcup m_{MustAccess}) \bigcap (n_{MayAccess} \bigcup n_{MustAccess}) \neq \emptyset$$

That is, two transactions are connected by an edge in this interference graph if both may access some particular shared variable.

3.3.1 Construction

The process for building the transaction interference graph for a given transactional program P is outlined in Algorithms 2-4. In the first step, the set of transactions in the program (or compilation unit in the case of incomplete programs or programs that use libraries) is discovered by the `DetectTxns()` procedure in Algorithm 2. This pass works by inspecting each IR instruction in the program's body, and if it this instruction is a function call to the transaction "begin" method (in TL2, this call is `TxStart`) then a new transaction has been discovered and it is added to the set $Txns$ of transactions. Therefore this pass performs a purely linear scan over the function's definition.

After the transactions have been discovered, the static Interference Graph INT for these transactions is initialized - each transaction is allocated a node in this graph and the edges which correspond to the interference relationship between these transactions are discovered in the following steps. The DS analysis described above computes the set of data structures that are referenced in a particular function and all its callees in a top-down fashion. However this set may be too conservative since we are only interested in the set of data structures that are accessed by transactional loads or stores. In this next step, this estimation is pruned by the `PopulateAndPrune()` procedure (Algorithm 2, line 7). This procedure is shown in Algorithm 3.

3.3.2 Pruning

The pruning pass takes the conservative set of DSNodes a transaction is expected to reference and refines it. For a transaction t it iterates over t's body and finds transactional loads and stores (which are implemented by the functions `TxLoad` and `TxStore` respectively in TL2). If an instruction being scanned is a call instruction with one of these two functions as targets, then we extract the pointer operand of this instruction which will be the pointer containing the address from/to which the load/store is being performed. The pass then attempts to map the abstract IR object corresponding to this pointer value to a node in the DS graph G. That is, it tries to find the exact data structure that is being referenced through this pointer. This DS graph node is then added to the *PrunedSet* for t. Otherwise if the instruction is a call to any other callee, the `PopulateAndPrune` pass is called for this callee. In this manner the subgraph of the call graph rooted at the parent function of t is processed in depth-first order.

After pruning has been performed, we have a less conservative estimation sets of DS graphs nodes that each transaction may reference. For two transactions $t1$ and $t2$, therefore their accesses to program data structures interfere if: (Algorithm 2 line 2)

$$PrunedSet(t1) \cap PrunedSet(t2) \neq \emptyset$$

Two transactions may have a conflict with each other through concurrent reads/writes through normal pointers that do not reference any data strcutures. To detect this type of interference in addition the the DS graph nodes, we also check if the pointer values in `PointerSet` corresponding to the pointer values referenced by transactions in load/store operations alias each other. More precisely, for two transactions $t1$ and $t2$ we check if each pair of pointer values $< p1, p2 >$ where $p1$ and $p2$ are pointer values occuring in $t1$ and $t2$ respectively, alias each other and if one of $p1$, $p2$ is used in a transactional store operation. If so, we say that $t1$ and $t2$

interfere with each other.

Algorithm 2 Algorithm for constructing the Interference Graph INT

Input: Program P

Output: Transaction Interference Graph G

```
1  DSInterferes (Txn t1, Txn t2) {
2  if PrunedSet(t1) ∩ PrunedSet(t2) == ∅ then
3      return FALSE
4  else
5      return TRUE
6  }
   DSAInterference (Program P) {
   TxnSet Txns = DetectTxns(P)
   INT = InitInterferenceGraph(Txns)
   foreach txn t in Txns do
7      Function F = t→Parent
       DSAGraph G = F→getDSAGraph()
       AddGraph(t, G)
       Processed ∪= {F}
       PopulateAndPrune(t, t→start, t→end)
8  foreach unique pair <t1,t2> ∈ Txns do
9      if DSInterferes(t1, t2) || AliasInterferes(t1, t2)
       addEdge(INT, t1, t2)
10 }
```

Algorithm 3 Algorithm for populating and pruning the DSA Node set

Input: Transaction T

Output: Pruned set of DSA nodes $PrunedSet(t)$

```
11 PopulateAndPrune (Txn t, Inst start, Inst end) {
   foreach Inst i ← start to end do
12     if CallInst c = isCallInst(i) then
13         target = c→getCalledFunction()
           if isTxLoad(target) || isTxStore(target)
           || isTxAlloc(target) || isTxFree(target) then
14             Value PtrValue = getPointerOperand(c)
               DSANode n = findDSANode(PtrValue)
               PrunedSet(t) ∪= {n}
               PointerSet(t) ∪= {PtrValue}
15         else
16             if !Processed.find(target) then
17                 Inst < fstart, fend > = target→getBoundaries()
                   Processed ∪= {target}
                   PopulateAndPrune (t, fstart, fend)
18 }
```

Algorithm 4 Check whether two transactions interfere through transactional loads or stores to alias pointers

Input: Transactions $t1$ and $t2$

Output: True if $t1$ and $t2$ interfere through aliased pointers

```
19 AliasInterferes (Txn t1, Txn t2) {
   foreach pair of pointers < p1, p2 > such that p1 ∈ t1→PointerSet and
   p2 ∈ t2→PointerSet do
20     result = DSAlias(p1, p2)
       if result = NoAlias then
21         return False
22     else
23         return True
24 return False
   }
```

3.4 Lock Allocation and Assignment

One fine-grained lock assignment scheme would be to map each memory location to its own abstract lock and each critical section that accessed this memory location would have that lock in its lock-set. Such a scheme would permit more concurrency but it would suffer from high lock acquisition/release overheads since for large transactions, the number of memory locations accessed may be quite large. On the other hand, allocating and assigning a lower number of locks means correspondingly lower lock acquisition/release overheads but also excessive serialization. Therefore the goals of minimizing the total number of locks assigned and increasing concurrency between critical sections are counter to each other. In [120] the authors provide a formulation for lock assignment that is optimal - it finds the lowest number of locks that can be allocated while maximizing concurrency. In this and the schemes proposed in [45, 121]

a total of between 1-3 locks were allocated and assigned which means much of the program execution is serialized.

Our baseline purely optimistic system uses word-granularity locks on memory locations to synchronize transactional accesses to those locations. Therefore each memory address *addr* has a non-unique word lock protecting it. To reduce the total number of locks usually each lock is mapped to multiple memory locations. Therefore for *addr*, the transactional word-lock protecting it can be computed by hashing the address and finding the appropriate index in a lock table [121]. We refer to these locks as *transaction word-locks*.

In our hybrid scheme we augment these transactional word-locks with a number of coarser grained locks derived from the interference graph analysis above. We refer to these locks as *assigned locks*. In addition we use a special *commit* lock to synchronize global commits. We use the transaction word-locks to synchronize concurrent accesses between *revocable* transactions and the assigned locks to synchronize accesses between concurrent irrevocable transactions as well between irrevocable transactions and revocable transactions. This concurrency control mechanism is described in detail below.

3.5 Runtime Support

3.5.1 TM Model

At a high-level, the TM model we consider is that of a *lock-based*, *write-back*, software TM that guarantees *opacity*, uses commit-time locking and performs validation at both encounter time (during an access) as well as at commit time. This describes a large variety of systems including TL2 [1], TinySTM [7] and DSTM [9] among others. A thread begins executing a transaction T by calling `tm_start()`. In this step all of T's data structures such as read/write sets, filters etc., are allocated and/or initialized. The global clock is also sampled and the timestamp is stored as T's start

time. This clock is simply a monotonically increasing global counter and the start time is used in the conflict detection stage for determining whether a variable accessed during execution of T was concurrently updated by another concurrent thread. The body of T the `tm_read()`, `tm_write()` and related calls for performing speculative accesses to shared data. When finished, T attempts to commit by calling `tm_end()`. This marks the start of the validation (also referred to as conflict detection) phase which we describe in more detail below.

3.5.2 Access & Commit Protocol for Revocable Transactions

Validation: In the first step then attempts to acquire the *commit* lock and blocks if it is already held. This lock ensures that T does not write to a memory location being read by an irrevocable transaction. After acquiring this lock, T attempts to validate $\mathsf{R}_{\mathsf{T}_p}$ and validate *and* acquire a lock on each element in its $\mathsf{RW}_{\mathsf{T}_p}$ and $\mathsf{W}_{\mathsf{T}_p}$ sets. The outline of this step for $\mathsf{RW}_{\mathsf{T}_p}$ is shown in Algorithm 1. For each element e in $\mathsf{RW}_{\mathsf{T}_p}$ its current version number is compared to T's start time. If the former is greater, then e was updated by another transaction i.e., e is invalid and T is aborted. If not, it checks whether e is currently locked by another concurrent transaction. If it is then the latter will most likely commit sometime in the future and update e thereby rendering T's copy invalid. Thus in this case too it aborts immediately. If e was both valid and not locked then T attempts to acquire a lock on it and aborts if it is not able to. This process is repeated for every element in its read-write and write sets. In the next step the read set for T is validated. This is similar to the above except no locks are acquired - for an element in the read set, if it is not currently locked and its version number is lower than T's start time then the element is considered valid. If all the elements of the read/write sets have been found to be valid and all the transaction locks are successfully acquired, then T is valid with respect to all other revocable

and irrevocable transactions. At this point T is considered to have been validated and it moves into the write-back stage. In this stage, the values computed by T and produced into its local write buffer are finally committed to main memory. After this, the transaction has finished committing and releases all the locks it acquired in the validation step above.

3.5.3 Access & Commit Protocol for Irrevocable Transactions

An irrevocable transaction begins by calling `tm_start()` like a revocable transaction. It then attempts to acquire teh *commit* locks and all the locks in its lock-set. These locks may be held by another concurrent irrevocable transaction in which case this transaction waits for them. Note that we enforce a linear order of acquisition of locks to prevent deadlocks. That is we assign an arbitrary ordering among the set of locks that were assigned during the interference analysis step. Each irrevocable transaction attempts to acquire these locks in this order.

A transactional read operation in an irrevocable transaction is essentially a *non-faulting* load operation to the specified address. A transactional write operation is similarly equivalent to a store operation except that the transaction also samples the global clock and updates the version number of the memory location being written to. When this transaction attempts to commit, all of its reads and writes are guaranteed to be valid (i.e., no other concurrent transaction has modified those memory locations) and the transaction commits by simply releasing all its *assigned locks* and the *commit* lock.

3.6 Experimental Evaluation

We implemented the compiler support for generating assigning and mapping locks in the LLVM [15] compiler (v2.4) and runtime support in the TL2 TM system [1]. In this section we analyze the performance impact of promoting transactions to be irrevocable through experiments on parallel transactional workloads in the STAMP suite [3]. The

Table 3: Description of programs & input sets. †=STAMP benchmark or library [3]

Program	**# Txns**	**Description**	**Contention**
list†	3	Lookups(85%), inserts(10%), deletes(5%) on a concurrent linked list	High
intruder†	3	-a10 -l128 -n262144 -s1	High
labyrinth†	3	-i inputs/random-x512-y512-z7-n512.txt	Low
kmeans†	3	-m15 -n15 -t0.00001 -i inputs/random-n65536-d32-c16.txt	High
ssca2†	5	-s20 -i1.0 -u1.0 -l3 -p3	Low
genome†	5	-g16384 -s64 -n16777216	Moderate
vacation†	3	-n4 -q60 -u90 -r1048576 -t4194304	Moderate
yada	6	-a15 -i inputs/ttimeu1000000.2	High

`list` program is a library component of STAMP that is used extensively in many workloads in the suite. The `counter` program implements a simple shared counter updated concurrently by several threads, a commonly occurring parallel programming artifact. We used an unmodified TL2 STM [3] as our baseline optimistic concurrency control system. Both the unmodified TL2 baseline and our Hybrid-Irrevocable TL2 STMs use write-buffering, lazy-validation and commit-time locking. All workloads were compiled using LLVM and gcc-4.3.3 for final code generation, with the default optimization flags for each workload. We ran all experiments in Linux on a machine with dual Intel Xeon X5500 4-core processors in which each was core clocked at 2.93GHz and each core also had hyperthreading enabled (for a total of 16 contexts). To reduce interference due to scheduling each thread was bound to a specific processor core uniformly. All the workloads were executed with the standard reference inputs if defined (else the inputs are described in the discussion below). The baseline versions of the programs use normal optimistic concurrency control in transactions using an *unmodified* TL2 STM and are shown as ``baseline'' in the plots. The Hybrid-Irrevocable STM versions of the program are shown as ``Irr'' in Figures 13-16.

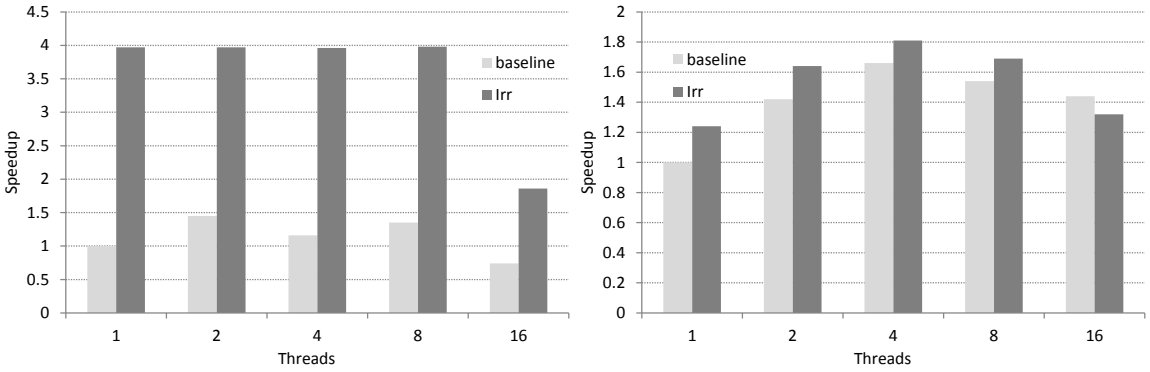


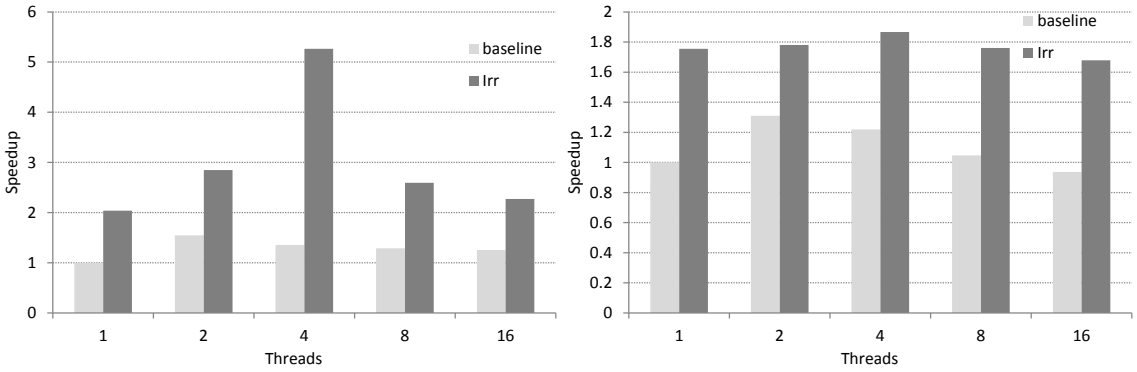
Figure 13: Parallel Speedup from our Hybrid Irrevocability scheme over single-threaded TL2 for (a) `list` (b) `genome`

list: The `list` program implements several linked lists each without duplicate key values. This program (or rather the linked list library used by this program) is used extensively in the other STAMP benchmarks. The program creates and initializes an initial set of lists and launches several threads which perform concurrent operations on them. An operation can be one of `insert`, `find` or `remove` with a specified key to insert, find or remove with each of them corresponding to 20%, 60% and 20% respectively of the total number of operations performed on each list. Each of these operations is implemented as a transaction. Given a key `k` to insert the `insert` routine iterates over a list and finds the right position to insert this key into. Then the actual modification of "`next`" pointers takes place as in standard list insertion. Similarly the `remove` routine iterates over a list to find the element to remove. The insert and remove routines also increment and decrement the `size` of the particular list. Since all three operations involve traversing through the nodes in a list, most of the time spent in transactions in this program is spent in reading the `next` pointers and comparing the key in a node to the given key. Moreover, the majority of the transactions in this program are quite large in terms of their read sets and hence the average amount of wasted work due to a conflict is very high. The improvement in parallel performance for this program from our hybrid irrevocability scheme is

significant - almost 3.3X as shown in Figure 13. The baseline version of this program has a very high level contention as evidenced by an abort rate of almost 74% for four threads. With our irrevocability scheme this is reduced to around 50.1%.

genome: The `genome` benchmark implements a gene sequencing program that reconstructs the gene sequence from segments of a larger gene. The program contains five transactions - two of which together account for a significant fraction of the total time spent in transactions. These transactions perform query and insert operations on a shared table data structure which is in turn backed by a concurrent linked list. Overall, all the dynamic transactions in this program are quite short and there is little contention among them - the abort rate in the TL2 baseline version of the program has a transaction abort rate of less than 0.01%. Consequently, the performance improvement due to promoting transactions to be irrevocable is small - about 1.17X for two threads as seen in Figure 13. From this figure we also see that the transactional overheads during single-threaded execution are not that high - about 1.25X which is an additional reason for the limited performance improvement seen.

kmeans: The `kmeans` program implements a transactional version of the popular Kmeans algorithm using optimistic concurrency control [3]. This workload contains a total of three critical sections implemented inside transactions. The first two add a value to shared scalar variables while the third (which is larger in size) atomically increments elements in a region of an array. This transaction accounts for most of the contention in this program and is consequently also the one that experiences the most number of aborts. Therefore the amount of work wasted due to this contention is quite high in the baseline TL2 version of the program. With our hybrid irrevocability scheme, this large transaction is frequently promoted to be irrevocable and we see that there is an improvement of almost 3.7X over the baseline TL2 version of the program for 4 threads as shown in Figure 14. From the same figure we also notice



(a) (b)

Figure 14: Parallel Speedup from our Hybrid Irrevocability scheme over single-threaded TL2 for (a) `kmeans` (b) `intruder`

that the transactional overheads for single threaded execution are also quite high for this program - for a single thread, the hybrid scheme performs almost 2X better than the baseline optimistic concurrency control scheme in TL2.

Intruder: The `intruder` program implements a signature based network intrusion detection system. The targets of contention for this program are several queue, list and tree datastructures that are used in the network packet capture, reassembly and detection phases. Much of the functionality is implemted in three transactions one of which does the bulk of the packet decoding. The amount of contention in this program is high owing to the frequency at which the packet reassembly phase rebalances its tree structure. The abort rate for the baseline program is around 14% for four threads. Moreover much of this contention occurs in the largest transaction in the program. By frequently promoting this particular transaction to be irrevocable, we see a speedup of over 1.78X for 16 threads as shown in Figure 14.

labyrinth The `labyrinth` in STAMP implements Lee's algorithm for finding the shortest distance between two given points on a grid. Most of the functionality in

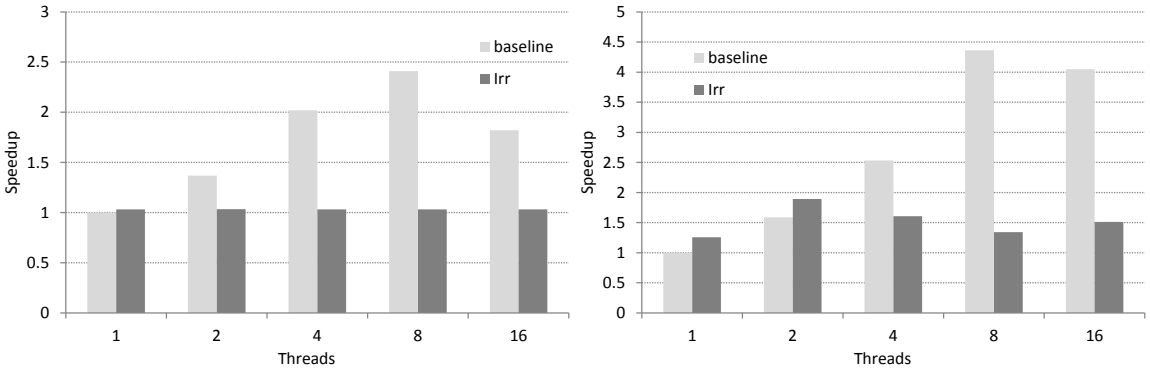


Figure 15: Parallel Speedup from our Hybrid Irrevocability scheme over single-threaded TL2 for (a) `labyrinth` (b) `ssca2`

this program is implemented within three transactions. The largest of these transactions which implements the bulk of the route finding algorithm, checks the status of a shared flag that denotes whether a particular point on the grid is already occupied by some other route. If this flag is set, the transaction forces itself to restart without even attempting to commit. This explicit *retry* is detected in our compiler scheme and this transaction is marked as not suitable for making irrevocable since then the transaction could not have a safe way of restarting. This means that only the other smaller transactions are considered for irrevocability thereby limiting the improvement in parallel performance. Adding to this, the amount of contention in the baseline program is also quite low - the abort rate is less than 0.01%. Therefore we do not see any improvement in performance over the TL2 version of the program, in fact we see a sginificant slowdown.

ssca2: Most of the critical sections in `ssca2` are small and perform simple operations such as increments or adding scalar values to shared variables. Most of the time spent in this program is spent in one particular critical section which is inside a 2-deep loop nest but is also quite small in terms of the sizes of the read and write sets of the transaction. Moreover, the low number of *assigned locks* generated in the

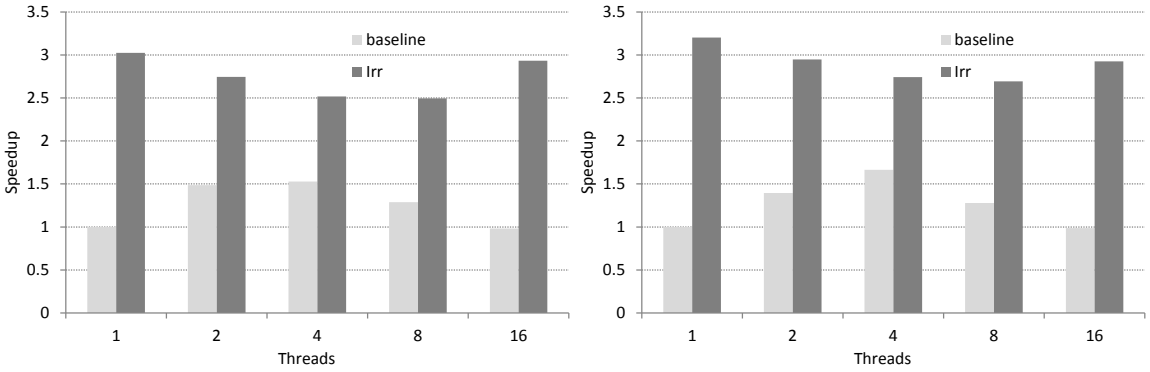


Figure 16: Parallel Speedup from our Hybrid Irrevocability scheme over single-threaded TL2 for (a) `vacation` (b) `yada`

interference analysis phase means that most of the execution within transactions is serialized despite the amount of dynamic contention in this program being quite low (the abort rate is ¡ 0.01%). As a result we do not see any improvement in parallel performance and in fact see a significant slowdown as seen in Figure 15.

vacation: The `vacation` program from STAMP implements a travel reservation system powered by a non-distributed database. The database consists of several tables which are implemented as Red-Black trees internally. The program implements three transactions one each corresponding to the three main actions - querying and adding reservations to the database, adding and deleting customers and updating the tables to add services or products that can be reserved. The abort rate for the baseline version of the program is low - about 0.6%. However the transactional overheads remain high as shown the by improvement in single-threaded performance of nearly 3X using our hybrid scheme. For multiple-threaded execution, the improvement in performance is significant - almost 3X for 16 threads (Figure 16.

yada: The `yada` benchmark implements Ruppert's algorithm for Delaunay mesh refinement. It consists of six transactions, one of which simply performs an atomic

add operation on two values. This program has a significant amount of contention - the baseline transactional version of the program has a 39.6% abort rate for 4 threads. Our hybrid scheme improves parallel performance substantially over the baseline - we see a maximum improvement of almost 2.9X for 16 threads as seen in Figure 16. Single threaded performance is nearly 3.2X better than the baseline indicating the high monitoring and validation overheads in this program even without aborts and wasted work.

3.6.1 Insights

Our experiments indicate that there is a small set of transaction characteristics that can be used to qualitatively predict whether promoting transactions to be irrevocable improves overall parallel performance of a particular program. Some of these are:

1. **Dynamic Size of Transactions:** Much of the overhead from optimism stems from the extensive monitoring and validation of transactional read/write accesses. This overhead is therefore correlated with the total dynamic number of transactional read/write accesses in a transaction - a metric that we refer to as the dynamic *size* of the transaction. Since an irrevocable transaction does not have much of this type of overhead, as expected, we have found that the larger the size of a transaction, the larger the improvement in parallel performance. In the case of the `list` & `intruder` programs this factor accounts for much of the improvements in parallel performance seen in Figures 13 and 14. On the other hand the very short transactions in `ssca2` (Figure 15) are not helped by irrevocability.

   The plot in Figure 17 shows the influence that dynamic transaction size has on the speedup due the hybrid irrevocability scheme. Programs that have large transactions show larger speedups compared to programs with small transactions.

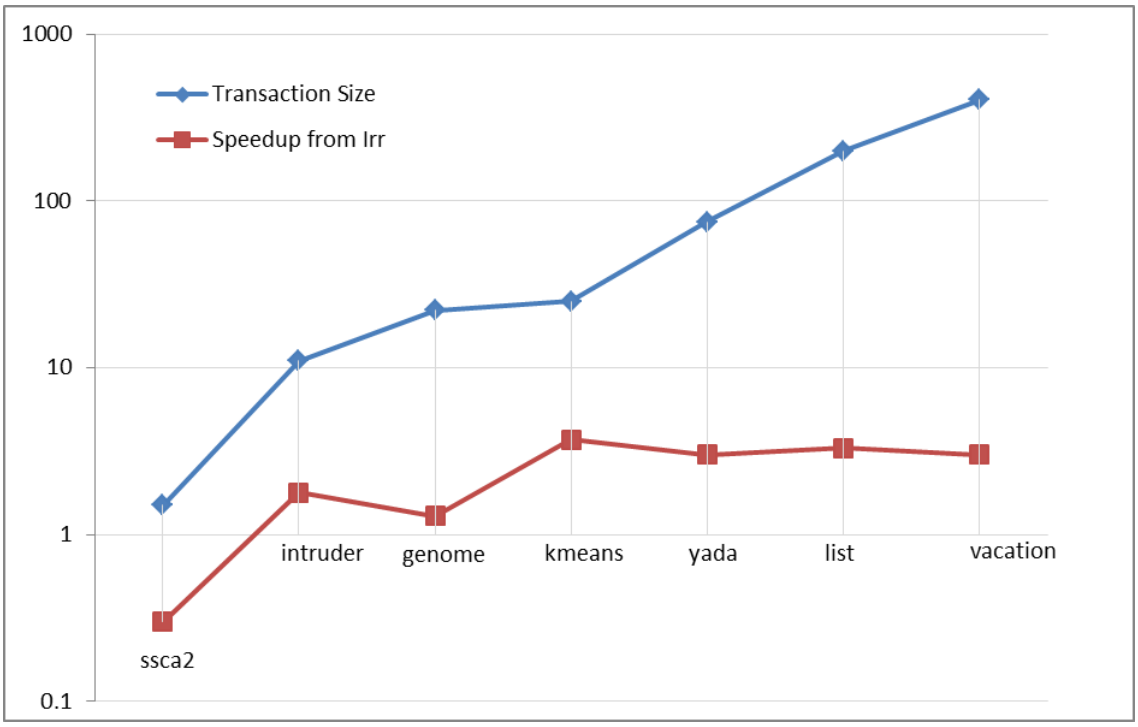


Figure 17: Plot showing the impact of dynamic transaction size on the speedup obtained for the STAMP suite. Workloads with larger average dynamic transactions size show higher maximum speedups

2. **Static Interference:** The interference graph for a program describes in a sense the amount of *static* contention in the program - i.e., the degree to which different transactions access the same program-level data structures. We found that the *density* of the interference graph plays an important role in the actual parallel speedup. A dense interference graph means that most of the transactions touch the same set of static data-structures (as per the conservative DS analysis) and hence these transactions cannot perform disjoint accesses. In this case, promoting transactions to be irrevocable has limited benefit since these irrevocable transactions will need to be serialized.

Table 4: Reduction in number of memory references due to *Irr*. All numbers are for 8 threads.

Program	**# Memory References (Baseline)**	**# Memory References (Irr)**	%Reduction
list	1249768705	1233562749	1.29
intruder	132710212586	51011172937	61.56
kmeans	43572344351	3113867235	92.85
ssca2	80691650050	72225407292	10.49
genome	11282223332	11210013629	0.64
vacation	211528183848	84014914936	60.28
yada	131561778243	79523450895	39.55

3. **Dynamic Contention:** Dynamic contention in a program in a particular time-interval corresponds to the frequency of two concurrent transactions accessing the same shared memory word at runtime during that time-interval. Our experiments indicate that promoting specific transactions to be irrevocable is profitable for high-contention programs whereas for low-contention programs the performance improvements are smaller. The reason is that high-contention programs typically have high abort rates. An abort in a revocable transaction means that it is forced to restart from scratch which means that it incurs the overheads inherent to transactional accesses once more. On the other hand in

an irrevocable transaction, this transaction would not have pay these overheads. Note however that the relationship between contention levels and the profitability of making particular transactions irrevocable is not straightforward because making a transaction irrevocable generally also tends to *increase* the amount of contention. This is because, in the presence of irrevocable transactions, normal revocable transactions contend with them for commit locks (see Section 3.5). However overall, in our experiments, we observed that the programs which were designated as "high contention" in Table 3.5.3 showed larger improvements in parallel performance.

4. **Abort Rate:** Like the contention metric described above, the abort rate in a normal transactional program (consisting of purely revocable transactions) is correlated to the magnitude improvement in parallel performance with our hybrid scheme. The `labyrinth` and `ssca2` programs (Figure 15) for example have very low abort rates to begin with (each ¡ 0.01%). getLock(e)

   The plot in Figure 18 shows the influence that dynamic contention and the abort ration have on speedups from the hybrid irrevocability scheme. Programs that have high dynamic contention and high abort ratios show larger speedups compared to programs with low abort rates.

5. **Dynamic Frequency of transactions:** We observed that for the programs in Table 3.5.3, the dynamic frequency of transactions was indicative of the improvement in parallel performance from promoting transactions to be irrevocable. This is expected since the frequency of transactions also indicates the amount of overhead being incurred during execution.

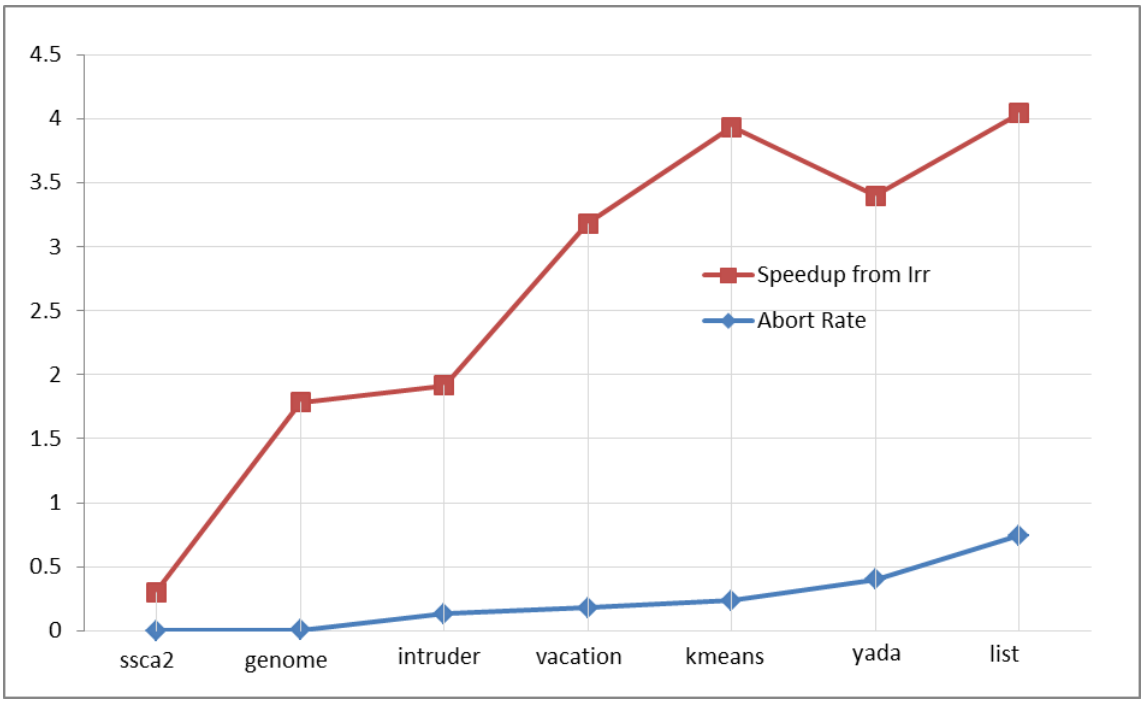


Figure 18: Plot showing the impact of dynamic contention on the speedup obtained for the STAMP suite. Workloads with high average abort rates show higher speedups

3.7 Conclusion

Irrevocability for memory transactions has so far been studied as a safety mechanism for guaranteeing correctness in the presence of unrecoverable operations such as I/O, exceptions or network operations inside transactions. In this work we have shown that conferring irrevocability on multiple concurrent transactions has very strong performance advantages. To ensure that these irrevocable transactions are synchornized correctly not only with each other but also with the normal revocable transactions we have built a hybrid concurrency control system that performs compile-time lock assignment using an interprocedural context sensitive data structure analysis for determining intereference relationships between transactions. Our experiments indicate this system improves parallel performance upto 3.3X relative to a normal TM system providing optimistic concurrency.

CHAPTER IV

VALUE-AWARE SYNCHRONIZATION

There is a large class of real-world programs termed *Soft Computing applications* [42] which are characterized by several unique properties.

- **Approximate nature of results.** These applications all produce an approximation of the actual results rather than their actual values. This may be because of several reasons. One common reason is that the physical or mathematical model expressed in the program requires some approximation to be computable in a reasonable amount of time. Other programs such as simulation applications mimic continuous processes but in a discrete-time fashion and this introduces some error in the result.

- **User-defined correctness.** In some cases, the application programmer can choose to consciously sacrifice accuracy of the results in order for the program to meet some execution characteristics such as soft real-time deadlines. He or she may be able to control parameters that directly determine the amount of error in the results produced. Examples of such parameters include thresholds in approximations, the granularity of ticks in time-stepped simulations, cutoff distances and radii in physical simulations etc.

- **Tolerance for Imprecision and Uncertainty.** Soft computing applications to some extent are tolerant of imprecision in inputs and some program values. Many such applications are designed to work with input streams and program values which are inherently noisy, imprecise or unreliable. Examples of such programs include pattern recognition systems, object-tracking systems and other

machine learning applications.

Several researchers have shown that for many such soft computing programs, it is possible to design optimizations that exploit these properties to improve performance by sacrificing the accuracy, precision or some other aspect of intermediate computations and of the result produced [44, 46]. In [47] the authors propose an FPU and architecture design that uses dynamic precision reduction for lower energy and area requirements. In this chapter we study the phenomenon of *store value locality* and its application to reducing synchronization conflicts in programs that use optimistic concurrency control such as hardware or software transactional memory systems.

4.0.1 Value-aware Synchronization

In a multithreaded program on a shared memory machine, shared variables are used to communicate values between different threads. This communication is synchronized using explicit constructs such as locks and mutexes or in the case of an optimistic synchronization system such as a hardware or software transactional memory system (H/STM) it is guaranteed by the runtime provided the programmer follows the constraints on specifying atomic sections correctly. For two concurrent threads, a write to a shared variable in one thread signifies production of a new value that may be consumed in the other thread. This production and consumption of values are usually synchronized precisely. However in many soft computing applications, the program may be tolerant of some level of imprecision in this synchronization. In the most common case, the consumer of a value from a shared variable may be able to proceed with its computation without receiving the newest value produced into that variable provided that the newest value produced is not too different from the old value that it read. That is, if consecutive updates made to the shared state are relatively small, then the consumer may be able to proceed with the older state without waiting for the newest value, as happens in normal (precise) synchronization. In the

following sections we show that for many programs a large fraction of dynamic writes update shared variable in this manner. We also show that this property combined with the properties of soft computing applications described previously allow us to reduce synchronization overheads and improve parallel execution performance.

The three major contributions of our work and the organization of the rest of the chapter are outlined below:

- We describe the phenomenon of *Approximate Store Value Locality* and show experimental evidence that establishes the existence of this phenomenon in many programs (Section 4.1).

- Given a similarity threshold, we propose a mechanism for detecting Approximate Store Value Locality efficiently in a program that uses optimistic synchronization (Section 4.5.1)

- We describe a technique to exploit this locality phenomenon in reducing the number of conflicts in several soft computing applications which are tolerant to imprecise sharing of data between threads (Section 4.5.2) and present an experimental evaluation of performance and accuracy (Section 4.6).

4.1 Approximate Store Value Locality

The phenomenon of *Store Value Locality* (SVL) in programs has been reported and studied widely in literature [11]. Briefly, a program is said to exhibit store (or shared) value locality when many write operations in the program write values that are either trivially predictable or exactly match the values already at the memory address being written. In this section, we show that a related but different property of *Approximate Store Value Locality* is also prevalent for many programs. This term describes the phenomenon where many writes write values that are *approximately local* to the values already at the memory address being written. We define *"approximate locality"* of

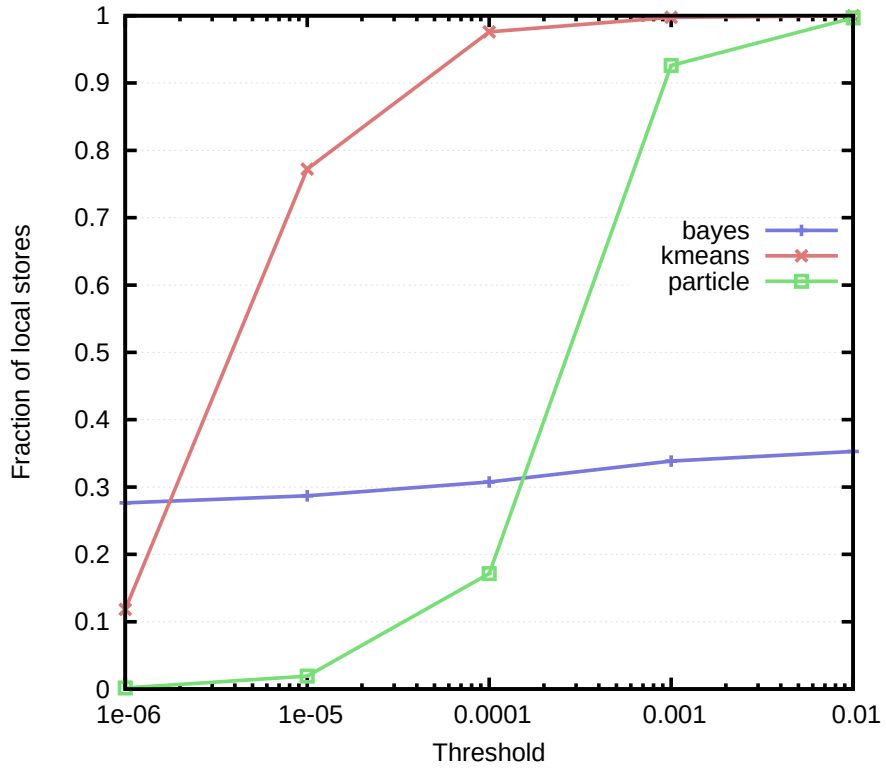


Figure 19: Approximate Shared Value Similarity in Critical Sections

two values v_0 and v_1 to be as follows:

" Two values v_0 and v_1 are approximately local for a small threshold τ if $\mid v_0 - v_1 \mid < \tau$ *"*

Therefore if a store instruction is about to write v_1 and the value v_0 is already present in memory at that address and the above condition is met, we say that the instruction exhibits *Approximate Store Value Locality* (ASVL) for the threshold τ and we call this store an *approximately-local store*. Whether a particular segment of code exhibits ASVL depends on the value of τ and the values themselves.

4.1.1 Approximate Value Locality in Critical Sections

In many real world applications, many of the values produced into shared variables in critical sections, undergo transformations that change them very little in relative terms. To test this hypothesis, we collected statistics on approximately value locality

for the programs shown in Figure 19. Specifically, we measured what percentage of stores to shared floating point variables inside transactional code committed values that were approximately similar to the values already present. The results are shown in Figure 19. In this graph the X-axis corresponds to the relative similarity between values written by stores to the same shared memory location. The Y-axis shows the percentage of total number of dynamic stores operations inside critical sections, that are exhibited this value similarity. We see that for all the programs shown, a substantial fraction of dynamic writes inside transactional code were *approximately local stores*. In these programs there are a lot of single or double precision floats and indeed in many cases most of the computation inside transactional code is performed on these floats - the number of approximately local stores that wrote integers was insignificantly low in all cases. These statistics tell us that a significant portion of shared values produced inside critical sections are arithmetically similar (the overall similarity being a function of the threshold). Since shared variables are typically used for communicating state or updates to state between threads, a related observation we can make is that for these programs,

"A significant portion of the values or updates being exchanged between the threads are relatively close to each other in magnitude"

In [11], the authors cite several reasons for the existence of store locality in real world programs. In addition to those factors, there are a few other empirical reasons that explain the ASVL phenomenon

- **Similarity in input data**: Many real-world input data sets contain a substantial number of input values that are similar.
- **Iterative refinement**: Many critical sections occur inside loops where the results computed in the loop body are synchronized with the global state at the end of each iteration. If the results computed are similar or approximately similar for two consecutive iterations (i.e., each thread, modifies global state by

a relatively small magnitude), then the store in the critical section that updates global state will often exhibit the ASVL property.

- **Finite Precision**: All real hardware has finite precision. Therefore knowing whether a *silent store* has occurred is itself an approximate endeavor if the store was writing a floating point value. Hence, for many programs which make heavy use of floating point numbers Store Value Locality manifests as Approximate Store Value Locality.

Most optimistic data synchronization mechanisms like transactional memories operate on meta-data such as versions and are oblivious to the actual values being shared between threads. Therefore systems with TMs, speculative lock-elision mechanisms etc., are unable to detect or exploit the approximate shared value similarity phenomenon. In Sections 4.2 4.5 we develop techniques to do both. While these techniques are discussed in the context of a TM system the broad principles apply to other optimistic synchronization systems as well.

4.2 Strong False-conflicts

In a transactional memory system, two concurrent transactions are said to conflict if both of them access the same shared variable and at least one of them performs a write operation on that variable. When such a conflict occurs, at least one of the transactions (usually the reader) is aborted. For example, consider the concurrent conflicting transactions T_1 and T_2 with the schedules below:

T_1(`start`); T_1(`write` v_1 in x); T_1(`commit`)

T_2(`start`); T_2(v_0 = `read` x); T_1(`commit`)

The TM system detects this conflict by determining whether the value read by T_2 could have been modified by T_1. Most TM systems typically use meta data such as version numbers with or without global clocks.

In TMs that use only version numbering, each shared variable or region of memory that can be accessed transactionally is associated with a version number. During a transactional read/write of this variable, this version number is cached by that transaction. A committing transaction increments the version numbers of all the variables that it is writing to (T_1 would increment the version number for x when it commits). During the commit phase, the version number cached for each variable read/written is compared to the latest committed version number for that variable. If the version number is the same, then there could not have been any writes to that variable since this transaction started. If the version number is different, then some other transaction must have written to this variable, and incremented its version number and a conflict is detected. Several other TM systems such as TL2 [1] additionally use the notion of global version-clocks, to order transaction start, read, write and commit events. In such systems, there is a global shared clock whose value (g) each new transaction reads when it starts. For each variable that can be accessed in a transaction there is a versioned write-lock (l). Each transaction also creates a local copy (wv) of the "write-version" by incrementing and fetching g. When a transaction wants to commit, it first iterates through its read and write sets to check if the corresponding l for that variable is less than g. If so, it is safe to commit. During the commit phase, the transaction iterates through its write set and for each variable therein, stores its new value from the write set and updates its versioned lock l to wv. In both types of systems described above and in general for most TM systems, a conflict for a shared variable is detected by comparing some local meta data for that variable with some global meta data. This method of detecting conflicts can result in pseudo-conflicts if the transaction commits the same or similar value as was present originally before the transaction started. Thus, if the concurrent transactions T_1(reader) and T_2(writer) have been found to conflict and T_2 commits the same value as existed when T_1 read it (i.e., the committing store operation in T_2 was a *silent store*), then we call this

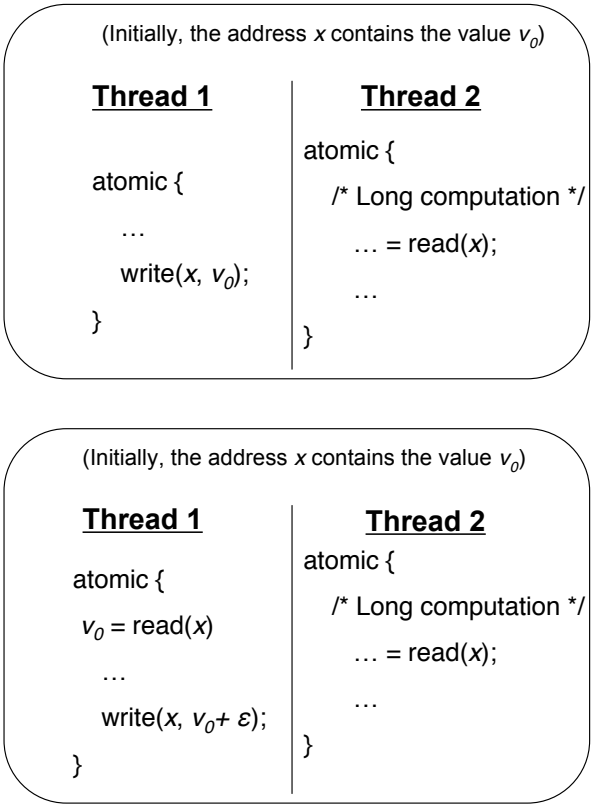


Figure 20: Example of two threads with Strong and Weak False-conflicts

conflict a *strong false-conflict*. Two distinct transaction schedules where this occurs are shown below:

$T_1(\texttt{start})$; $T_2(\texttt{start})$; $T_1(v_0 = \texttt{read } x)$; $T_2(\texttt{write } v_0 \text{ in } x)$; $T_2(\texttt{commit})$; $T_1(\texttt{commit})$

$T_1(\texttt{start})$; $T_1(v_0 = \texttt{read } x)$; $T_2(\texttt{start})$; $T_2(\texttt{write } v_0 \text{ in } x)$; $T_2(\texttt{commit})$; $T_1(\texttt{commit})$

We call these conflicts "strong false-conflicts" because ignoring them during the conflict resolution phase would not affect the correctness of the program. Redundant store operations such as the one in T_2 above can be eliminated by traditional compiler optimizations if the compiler is able to assert that v_0 is already the value at address x. However, this ability is restricted by procedure calls, indirect branches and other conditions in which the compiler cannot guarantee this condition is met.

4.3 Weak False-conflicts

The definition of "false conflict" requires us to define clearly what "equivalence" is. Determining equivalence is straightforward for data types such as integers and fixed point values, but is not well-defined for single or double precision floating point values since such values are represented with quantities with finite precision on any hardware. For floating point variables, we can only assert whether the two differ by at most some given value. This approximate equality is routinely used to compare floating point values in programs where the threshold for two floats to be considered equal, is supplied by the programmer. We call two floating point values to be "similar for threshold τ" if the difference between these values is smaller than τ.

We now extend the notion of false conflicts to include those caused by writing a value that was within some threshold of the original value that existed at that memory address. Therefore *for two concurrent transactions T_1 and T_2 accessing a shared variable x, if the value v_0 read by T_1 is overwritten with v_1 by transaction T_2 before T_1 commits and v_0 and v_1 are approximately-local for a threshold τ, then T_1 and T_2 are said to have a weak false conflict for τ*

That is, for the following transaction schedule:

T_1(`start`); $T_1(v_0 =$ `read` $x)$; T_2(`start`); T_2(`write` v_1 in x); T_2(`commit`); T_1(`commit`)

if $\mid v_0 - v_1 \mid < \tau$ then this schedule has a weak false conflict. This notion of similarity for a given threshold is well defined for native data types such as signed and unsigned integers and floating point values and fixed point values.

These false-conflicts arise because most widely used conflict detection mechanisms do not take into account the actual values being read and written and instead only use versions or such meta data. On the other hand, threads and the atomic units inside them care about values and not version numbers. Specifically, a reader of a shared variable does not usually care whether a particular variable has a different version

number provided it has the same (or in some cases, approximately similar) value. This would suggest that employing a conflict detection and resolution mechanism that had the ability to inspect the actual values of shared variables would improve concurrency at the cost of physical serializability.

4.4 Specifying Imprecise Sharing

Many real-world programs use thresholds for program quantities that results in an approximate final answer (often to improve execution time). Examples of such thresholds are cutoff radii in particle or molecular dynamics simulations, thresholds for scores in pattern matching and object recognition programs, timestamp windows for event processing in discrete event simulation systems and so on. In other programs, the programmer implicitly specifies this by having controlled, deliberately lazy updates to global shared data (for example in the Bayesian network simulation in [12]) and deliberate uses of stale, unsynchronized global state (many real-time particle systems such as in video games), among others. In all of these examples, there is some implicit or explicit specification by the programmer regarding what set of conditions can lead to an approximate but acceptable final answer. In our system, this specification is based on assertions about the *similarity* between values. Specifically, the programmer can use the threshold τ for specifying these assertions, knowing that given a program value a and its threshold τ_a, any value within τ_a distance of a is treated as semantically equivalent to a. All the implicit and explicit specifications for program approximations described above can be captured using this form.

4.4.1 Choice of Comparison Functions

In the previous section we defined the notion of weak false-conflicts for a particular threshold τ due to *approximate locality*. Determining approximate locality requires a robust and well-defined *Thresholded-Comparison* operation. In the simplest case described above this operation was simply an absolute comparison function and the τ

was also absolute. However this is inadequate since expressing absolute thresholds for changes in program values requires the programmer to be aware of the magnitudes of initial, final and intermediate program values. Instead our system uses the following comparison operations that use relative thresholds.

- $RelativeError(a, b, \tau_r)$: This operation determines if the *relative error* between values a and b is lower than the τ_r which is simply a float that describes the maximum permissible relative error. One well known problem with this operation is that it fails for numbers very close to zero. The positive float (double) closest to zero and the negative float (double) closest to zero are very close to each other but this function will determine them to be very far apart. However for other values this operation is fairly robust and intuitive to use.

- $MaxValues(a, b, \tau_u)$: This operation determines if the number of *representable* values between a and b is less than τ_u which is an integer. Therefore if this operation returns "true" for two floats a and b for $\tau_u = 1000$, this means that there are at most 1000 representable floats between a and b. This operation is more robust than $RelativeError$ but it requires reasoning about thresholds in terms of number of representable values between two program values.

4.4.2 Thresholded Types

The comparison functions described above are sound for integers, single and double precision floats. We define a set of augmented types that extend the native types with a threshold and a comparison function. These types are shown in Figure 21. Here `REL` and `MV` refer to the $RelativeError$ and $MaxValues$ functions respectively.

Scope: While avoiding weak false-conflicts may improve performance in many cases improper use of the thresholded types can inject error into the computation that renders the outputs meaningless or worse results in catastrophic failure of the program. Below, we list some important considerations in using thresholds for shared

data:

1. Smoothly changing values: The shared value to which a threshold is being applied should change smoothly relative to the threshold. Otherwise the consumer thread may observe values that change drastically.

2. Flag variables and predicates: Flag variables and predicates should not be thresholded as this will result in control flow being drastically changed.

3. Pointers: While the notion of a thresholded pointer may be useful in some cases, pointers should not be thresholded. This is because calls to functions such as `realloc` may leave the value in the pointer intact, but may change the attributes of the data or buffer being pointed to. Only native signed/unsigned integers, single/double precision floats should be thresholded.

4. Invariants: Programmatic or algorithmic invariants can be very useful in determining or controlling the amount of error introduced by using thresholded types. For example in a physical simulation for a closed particle system (discussed in detail in Section 4.6) the total energy in the system is a physical invariant (due to the first law of thermodynamics). Therefore the amount of tolerable error can be specified as a function of deviation allowed from this invariant and the thresholds can be determined accordingly.

5. Knowledge of program behavior: Finally, the value of the threshold for a particular variable must be based on the programmer's knowledge of the system being modeled and of the magnitudes of the quantities being computed. Just like STM features such as *partial commits*, *early release* and other mechanisms that affect the serializability and/or the correctness of the program, these imprecise synchronization techniques should only be used by expert programmers in situations when the implications are clear.

```
// Types using the "REL" function
typedef float(REL, 0.0001) RELThreshFloat
typedef double(REL, 0.0000001) RELThreshDouble
RELThreshFloat x;
RELThreshDouble z;

// Types using the "MV" function
typedef float(MV, 1000) MVThreshFloat
typedef double(MV, 1000000) MVThreshDouble
MVThreshFloat a;
MVThreshDouble c;
```

Figure 21: Extensions to native types for specifying thresholds and comparison functions

These thresholds are bound to the variables over the scope of the transaction and they will be used during the conflict detection phase described below.

4.5 Avoiding Strong and Weak False-conflicts

We defined a false conflict as one resulting from a silent or an approximately silent store operation in a thread that overwrote a value in memory with the same value or with a value that differed from it by a small τ. Certainly, in the case where τ is exactly equal to 0, (i.e., the store operation wrote the exact same value as the one already existing at that address), the conflict is not a real conflict since although ignoring it would affect the physical serializability of the transaction schedule, it would not affect the semantics of either transaction or those of the values produced therein. Therefore eliminating these conflicts would reduce transaction abort rate and therefore overall transactional throughput. Even for non-zero values of τ, many programs (or transactions in programs) can tolerate this approximate sharing of values and hence we would like to avoid weak false conflicts for a given τ. To do this we need efficient methods for two tasks - detecting approximate store value locality and avoiding conflicts that result from the occurrence of this locality.

4.5.1 Detecting Approximately-Local Stores

There has been substantial amount of work on detecting silent stores efficiently [11, 22] and a majority of them are based on program profiling and/or special purpose hardware for tracking stores. These works are discussed in Section 4.7. In our system however, instead of tracking silent stores, we want to detect Approximately Store Value Locality, i.e., store instructions that write a value that is within some small τ of the value already present at the address being written to. Moreover we would like to do this without the aid of profiling or special hardware. Fortunately, the use of optimistic synchronization (as in a TM system) for critical sections gives us a channel to monitor store values dynamically with little additional cost.

We will describe the detection technique in the context of a TM system like TL2 [1]. In TL2, there is a global version-clock variable g that is read and written by each writing transaction and is read by each read-only transaction. In addition each transacted memory location has a versioned write lock l that consists of a 1-bit *write-lock* predicate that indicates whether the lock is currently held by some thread and a *lock version* field that indicates the version number of the variable at that instant. Also, each transaction also has a table containing mappings from shared variables accessed in that transaction to their respective τ values that were specified by the programmer in the original program. At commit time a transaction attempts to acquire write locks for each of the elements in its write-set. If it is successful it will then perform an atomic increment-and-fetch operation on the value of g and record the returned value in a *local write-version number* variable wv. This value of wv is essentially the version number that this transaction will give to all the variables in its write-set after it has written to them. Then validation of the read-set is performed. If this is successful, a write-back is performed where for each variable a in the write-set, its new value buffered in the write-set is written to memory. Just before this write happens, we can determine whether the new value that is about to be written and

the old value at that address are *approximately local* for a threshold value τ_a. If so, we can simply mark this variable in the write-set as having been written to by a store exhibiting approximate store value locality. This step is shown in Algorithm 5 which is implemented in the commit protocol for the TM.

The cost of both the *RelativeError* and *MaxValues* comparison functions *is independent of the magnitude of the values being compared and is of the order of tens of instructions per comparison.* This is important for transactions that have large read/write sets and which therefore may invoke these functions frequently.

Algorithm 5 Detecting *Approximate Store Value Locality*

```
Require: Transaction T
  // Transaction writeback
  for all e ∈ WriteSet do
    if computeSimilarity(e.newVal, e.oldVal) == true then
      e.similarityflag = 1
    end if
  end for
  // Drop Locks
  for all e ∈ WriteSet do
    if e.similarityflag ≠ 1 then
      e.version = T.wv
    else
      e.version = T.rdv
    end if
  end for
```

4.5.2 Avoiding Conflicts due to Approximately-Local Stores

After a committing transaction has finished writing back the new values it has produced, it releases the write locks it holds and then clears the $write - lock$ bit for each variable in its write-set. The process of releasing a write lock for an element in the transaction's write set essentially consists of setting the *lock version* for that memory location to the local write version wv recorded in the transaction when it started its attempt to commit. This signifies a new value as having been produced and committed in the system.

In the detection phase described in Section 4.5.1 above, we identified and marked all variables in a committing transaction which were written to by an approximately-local store. Therefore while releasing locks for each variable in the write set in the current phase, we check to see if this variable was marked. If it was not, then the lock is released normally. If it was marked, then we bypass the updating the *lock version* for that variable to wv. Hence this variable contains the same version number as it did before this transaction acquired a lock on it. This is all the committing transaction needs to do.

A transactional read of a variable proceeds as follows. Before the load for the variable is executed, two other load instructions are first executed. The first one checks if the 1-bit write-lock is free. If it is set, then some other transaction is currently writing to this location and the transaction fails. If it is not set, then the *lock version* field wv is checked to make sure that it is lower than the transaction's read version rv. If it is greater than rv, then some other transaction has committed to it after the current transaction started. In the detection phase above, the wv field is not updated if the committing store is found to be *approximately local*. To see why this technique reduces conflicts, consider the example of a reader transaction T_2 in Thread 2 and a concurrent writer transaction T_1 in Thread 1 both accessing a variable x as shown in Figure 20. Let us assume T_2 started first. It first read the *global version clock* $g(= 0)$ into the thread local read-version variable rv_2. It then reads the value in x then proceeds to perform some computation. Sometime after that transaction T_1 starts and records the value $g(= 0)$ in its local read-version variable rv_1. It then executes an *approximately-local* store to x (which updates the value for x in its write-set). The transaction T_1 then attempts to commit. It updates g from 0 to 1 and sets its own wv_1 to 0 and then attempts to write its write-set to memory.

During this step, the ASVL detection mechanism described above is invoked and it marks the x in T_1's write-set as approximately local. The new value $(v_0 + \epsilon)$ is

then written to memory. Then T_1 attempts to release the write-lock on x and here the conflict avoidance mechanism described above is invoked. T_1 checks if the x in its write set is marked. Since it is marked, the *lock version* for x is *not* updated to wv_1 and is left unchanged at 0. Then if T_2 tries to commit, it checks to see if the *lock version* $\leq rv_2$. This will turn out to be true, since *lock version* $= rv_2 = 0$. Now T_2 can proceed to commit successfully.

This example describes one scenario and several others are possible with different orderings of transaction starts, reads, writes and commits.

4.6 Experimental evaluation

4.6.1 Experimental Setup

We implemented the false-conflict detection and avoidance techniques described above in the TL2 STM system. In this section we present results of our experimental evaluation of the techniques along two dimensions. Firstly we evaluated the effectiveness of our techniques in reducing the number of false conflicts and the aborts caused due to them. Secondly, we studied the amount and nature of error introduced in the program and the trade-off between accuracy and performance. We present these results below using case studies of three well-known parallel programs that can be characterized as *soft computing applications* according to the criteria outlined in the beginning of this chapter. The programs are `bayes` and `kmeans` from STAMP and `particle`. All programs were compiled with gcc-4.2 and executed on an machine with an Intel Quad Core processor with 4 hyperthreaded cores, each with an 8K L1 cache and 128K L2 cache running Ubuntu Linux. All running times were gathered using the gettimeofday() call. To minimize the interference due to system thread scheduling each thread was statically bound to a specific core. In all the experiments discussed below thresholds were applied only for single and double precision float types.

4.6.2 Case Studies

4.6.2.1 `Bayes`

A Bayesian network [12] is a way of representing probability distributions for a set of variables in a concise and comprehensible graphical manner. A Bayesian network is represented as a directed acyclic graph where each node represents a variable and each edge represents a conditional dependence. By recording the conditional independences among variables (the lack of an edge between two variables implies conditional independence) a Bayesian network is able to compactly represent all of the probability distributions.

Bayesian networks have a variety of applications and are used for modeling knowledge in domains such as medical systems, image processing, and decision support systems. For example a Bayesian network can be used to calculate the probability of a patient having a specific disease given the absence or presence of certain symptoms.

Algorithm 6 `Bayes`

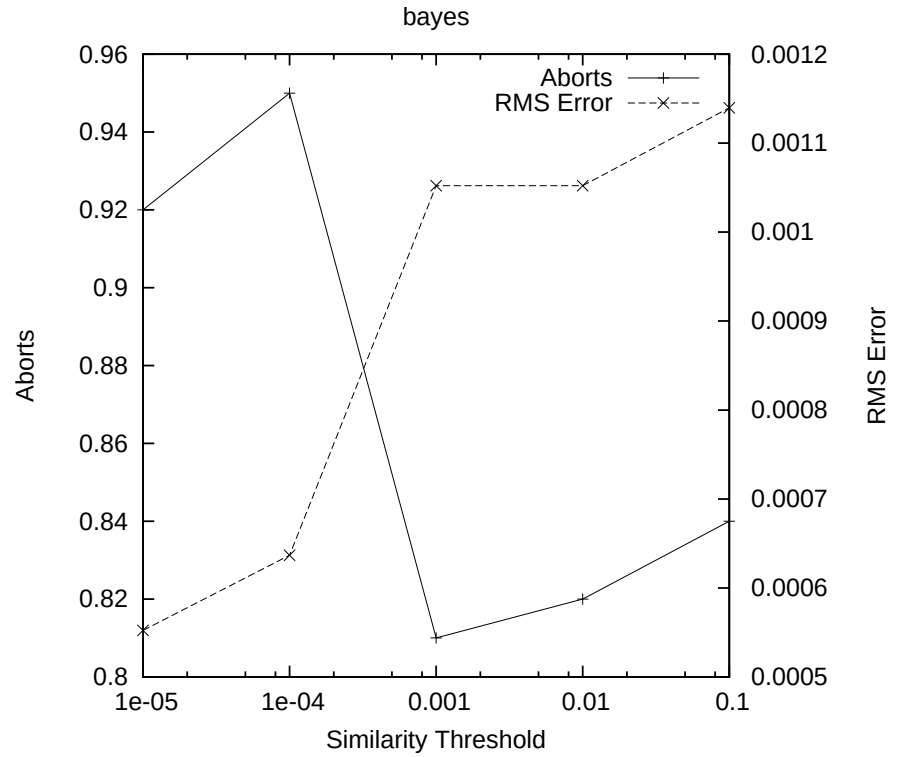
```
while (task = popTask()) ≠ NULL do
  if task.op→isInsert() then
    toID = task.toID
    newbll = computeLocalbll(toID)
    atomic {
    t = tm_read(localbll[toID])
    d += t - newbll
    tm_write(localbll[toID], n)
    } endatomic
  end if
  atomic {
  oldbll = tm_read(g_bll)
  newbll = oldbll + d
  tm_write(g_bll, newbll)
  } endatomic
  findAndInsertNextTask()
end while
```

This application implements an algorithm for learning Bayesian networks from observed data. The algorithm implements a hill-climbing strategy that uses both

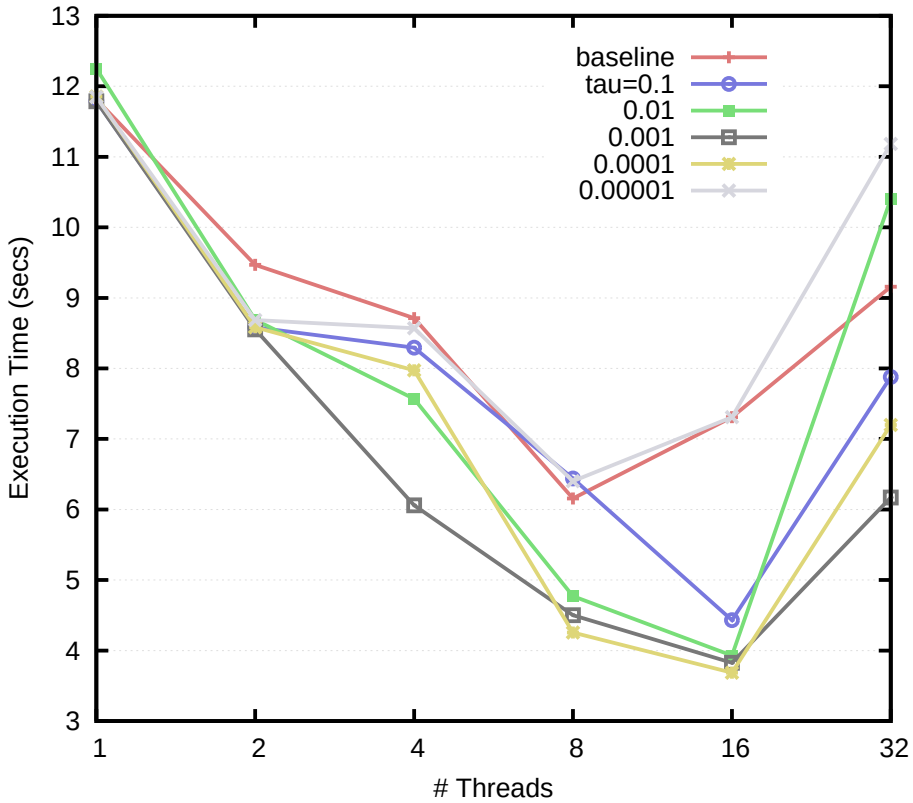
local and global search. The broad outline of the algorithm is shown in Algorithm 6. The network starts out with no dependencies between variables and the algorithm incrementally learns dependencies by by analyzing the observed data. On each iteration each thread is given a variable to analyze and as more dependencies are added to the network connected subgraphs of dependent variables are formed. A transaction is used to protect the calculation and addition of a new dependency, as the result depends on the extent of the subgraph that contains the variable being analyzed.

Computation of total base log likelihood The global base log likelihood (*g_bll* in Algorithm 6) is computed by computing the local base log likelihood for each variable, accumulating it, and finally atomically incrementing the current global log likelihood with this accumulated value. The application already implements an approximation wherein local log likelihoods are not communicated across threads to improve performance. We extend this by specifying a threshold τ for which the store of the global log likelihood can be considered *approximately similar* if the current global log likelihood is within that threshold. The τ is relative and so we use the *RelativeError* operation.

We show the amount of error and number of aborts for several threshold values in Figure 22a. The X-axis represents the thresholds used on a logscale. The left Y-axis shows the abort rate and the right Y-axis shows the corresponding amount of error in the result. The `bayes` program computes a learned score that it has learned from the observed data in addition to an actual score. We computed the amount of error in the learn scores produced by the program relative to the baseline and normalized this difference with the difference between lent and actual scores in the baseline case (which is the original program running with 4 threads). We see from the Figure 19 that a significant portion of dynamic stores are approximately similar for `bayes`. From Figure 22a we see that the number of aborts is reduced by almost 19%



(a) Error Vs. Aborts with REL threshold



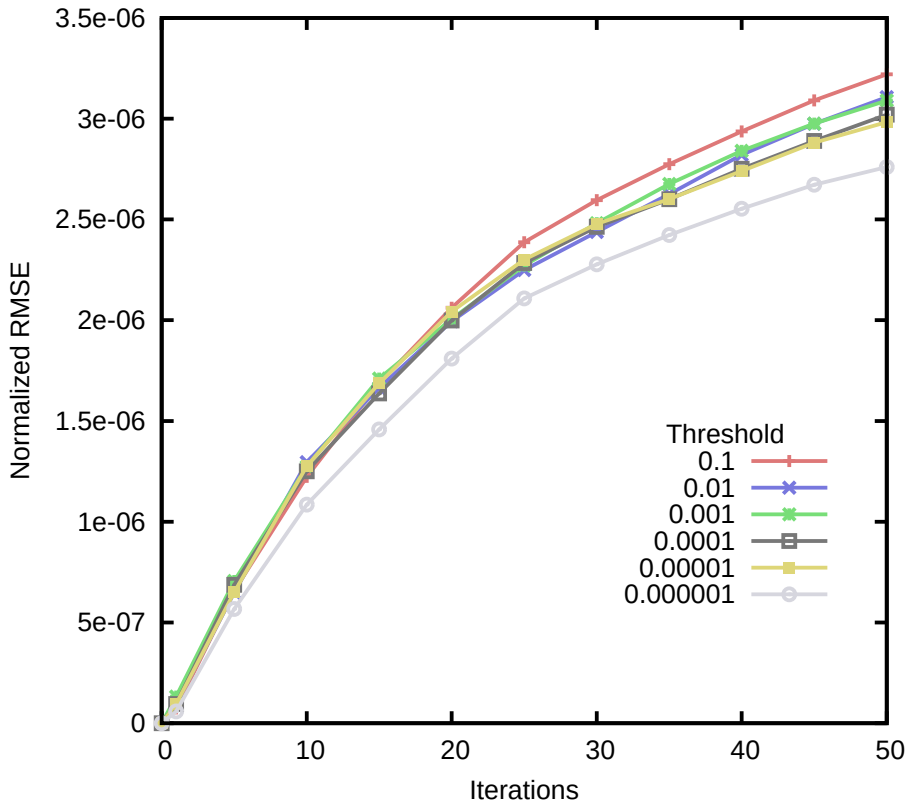
(b) Execution time with REL threshold

Figure 22: bayes

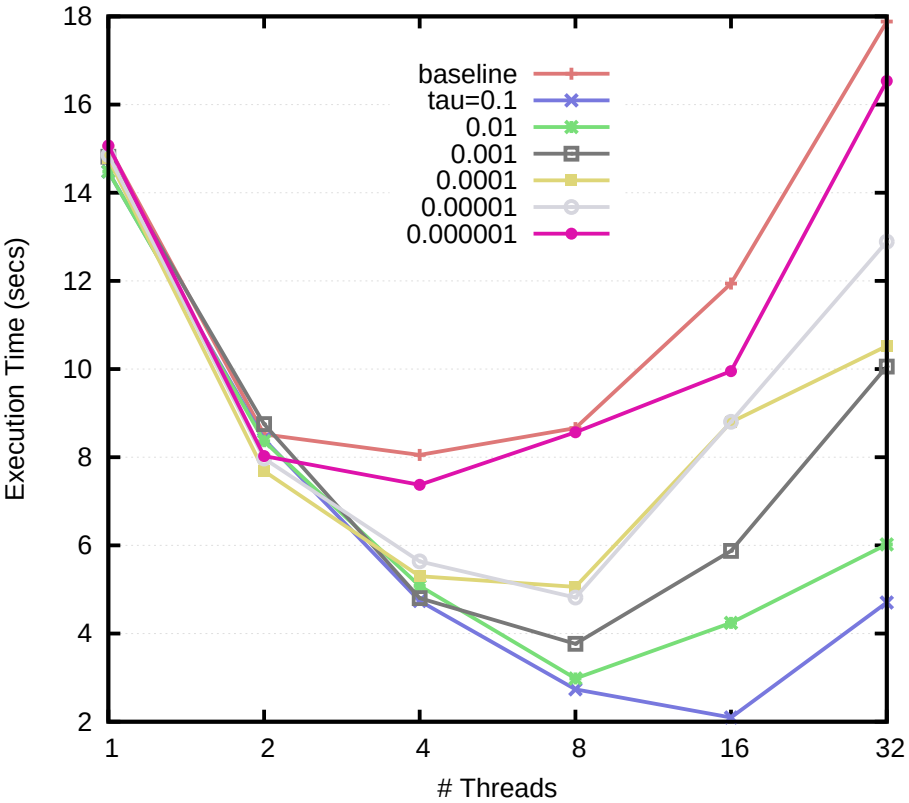
for a threshold of 0.001 producing a final error of roughly 5.3E-4. The calculation of new dependencies take up most of the execution time in this application causing it to spend almost all its execution time in long transactions that have large read and write sets. This program also has a high amount of contention as the subgraphs change frequently. Therefore by alleviating some of this contention through imprecise synchronization we are able to reduce the number of aborts which would in turn lead to improved execution time (since long running transactions implies a high penalty for aborting them). **One important property of this program is that the number of aborts and execution time depend on the order in which edges are inserted into the graph**. Therefore we expect the speedups as shown in Figure 22b to not be as smooth (see [12]).

4.6.2.2 `Kmeans`

The K-means algorithm [12] is a partition-based method to group objects in an N-dimensional space into K Clusters. It is commonly used to partition data items into related subsets, a common operation in many data mining applications. K-means represents a cluster by the mean value of all objects contained in it. The `kmeans` program in STAMP implements the K-means algorithm that is shown in Algorithm 7. Given the user-provided parameter k the initial k cluster centers are randomly selected from the database. Then each thread in the program is given a partition of the objects which it processes iteratively. Processing an object essentially consists of assigning the object to its nearest cluster center according to a similarity function. The Euclidean distance between the object and the cluster center is used as a similarity function. Once all objects in a partition have been processed new cluster centers are found by finding the mean of all the objects in each cluster. This process is repeated until two consecutive iterations generate similar cluster assignments i.e., there is no further reassignment of objects from one cluster center to another. The



(a) Growth of error with REL threshold



(b) Execution time with REL threshold

Figure 23: kmeans

TM version of K-means adds a transaction to protect the update of the cluster center that occurs during each iteration. The amount of contention among threads depends on the value of K. When updating the cluster centers the size of the transaction is proportional to the dimensionality of the space. Thus, the sizes of the transactions in `kmeans` are relatively small and so are its read and write sets. A conflict typically happens when a thread reads a cluster center for computing the distance from an object and another thread writes a new value for that cluster center.

Algorithm 7 `Kmeans`

```
while delta > 0 do
  delta = 0
  for all Object "i" do
    atomic {
    cc = findNearestClusterCenter(i)
    } endatomic
    if membership[i] ≠ cc then
      membership[i] = cc
      delta += 1
    end if
  end for
  for all Cluster "c" do
    atomic {
    c→center = computeNewCenter(c)
    } endatomic
  end for
end while
```

Computation of Cluster Centers. The cluster centers are computed by summing the objects within each cluster. These centers are computed and stored atomically in a transaction as shown in Algorithm 7. In the next iteration the distance of each point in a partition from all the cluster centers is computed. For a random distribution of initial cluster centers and objects, the relative amount of change in the position of the cluster centers is quite small over successive iterations. Therefore, we can apply an approximate locality threshold τ to the shared variables holding the positions of the cluster centers. Consider a thread A that has read the position of a particular

cluster center in order to compute its distance from objects in A's partition. Now the thread B that owns this cluster center computes a new cluster center which may be less than τ away from the current cluster center that A has read. Therefore the store executed by B is an *approximately local* store and would be marked as such. When thread A finishes computing the distances of each of its objects from the old cluster center these distances may be inconsistent. However if the relative magnitude of this inconsistency is small A can go ahead with the next step of reassigning objects instead of aborting and restarting.

We see from Figure 19 that a substantial portion of the values computed for cluster centers are approximately local. Figure 23 shows the error and performance characteristics for this program with a relative threshold and the `REL` comparision operation. From Figure 23a we notice that for a relative threshold of roughly 0.0001 an error of about 3E-6 was introduced. To compute the error introduced in the computation we calculate the root mean square error *RMSE* across all dimensions of all the points in the space. This RMSE is then normalized with the size of the space (which is the magnitude of the distance between the farthest points). The normalized RMSE is remains relatively small and grows smoothly across iterations during the execution of the program as shown in the Figure.

In this program, the amount of contention among threads depends on the value of K, with larger values resulting in less conflicts as it is less likely that two threads are concurrently operating on the same cluster center. However, even with large values of K, simply increasing the data set size (the number of points) increases contention among threads [12] and this effect was very apparent in our experiments. Therefore even though the algorithm was designed to be a low-contention one, the actual contention was quite high and consequently our relaxation technique produces significant improvement in transaction success rate. **Furthermore our experiments showed that the errors in final output for this program are comparable (around**

30-50% more than) to the RMSE in the outputs between several different runs of the *baseline* versions themselves. The plot in Figure 23b shows the speedup in execution time using the `REL` comparison operation with τ ranging from 1E-4 to 1E-6. A maximum speedup of roughly 5.7x is achieved for this program with 16 threads.

4.6.2.3 particle

Particle system simulations model the evolution of complex structure and motion of particles in a given system from a relatively small set of rules [123]. Such systems have been used in diverse scenarios ranging from stochastic modeling, molecular physics to real-time simulation and computer gaming. Particle systems have also been widely studied in the context of parallelization.

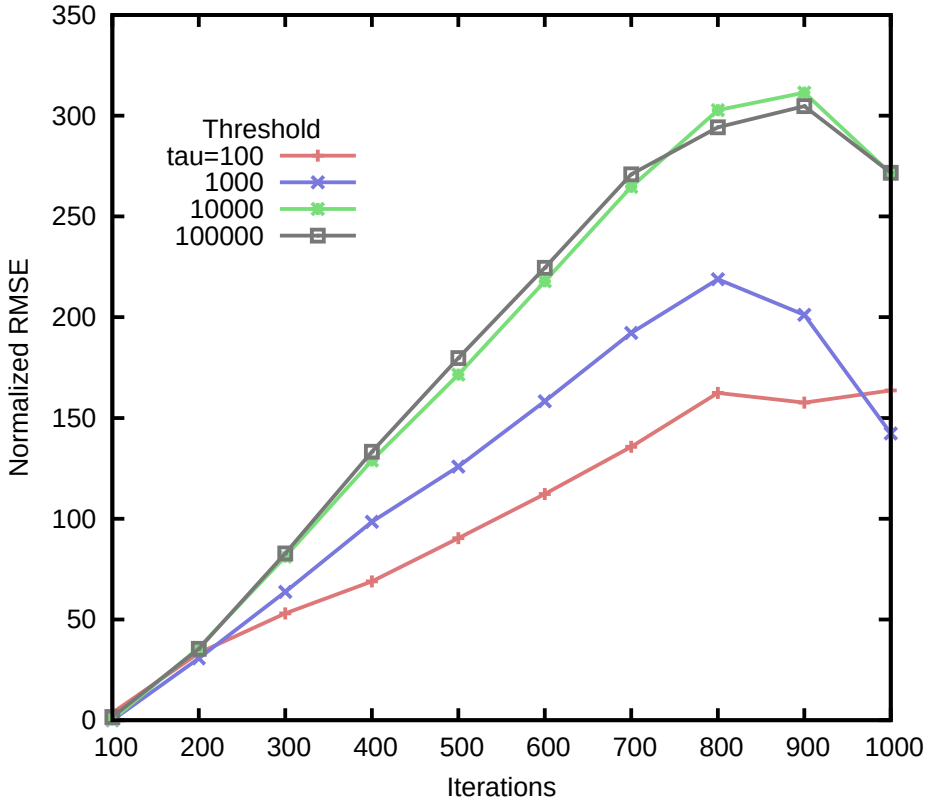
The specific particle system we describe here is similar to the one discussed in [123]. It consists of a number of particles distributed among a number of threads with each thread processing a distinct block of particles. Each particle has a position vector, a velocity vector and a mass associated with it. Each of the particles experiences two forces - a constant force (such as gravity) and also the gravitational force between pairs of particles. The system evolves in time-steps and, at each time-step, the movement of the particles due to these forces is computed using numerical integration methods. The outline of the algorithm is shown in Algorithm 8. The algorithm uses Euler integration to calculate the values of the position and velocity attributes of a particle p using the following equation

$$f_p\left(t+dt\right) = f_p\left(t\right) + dt * f_p^{'}\left(t\right)$$

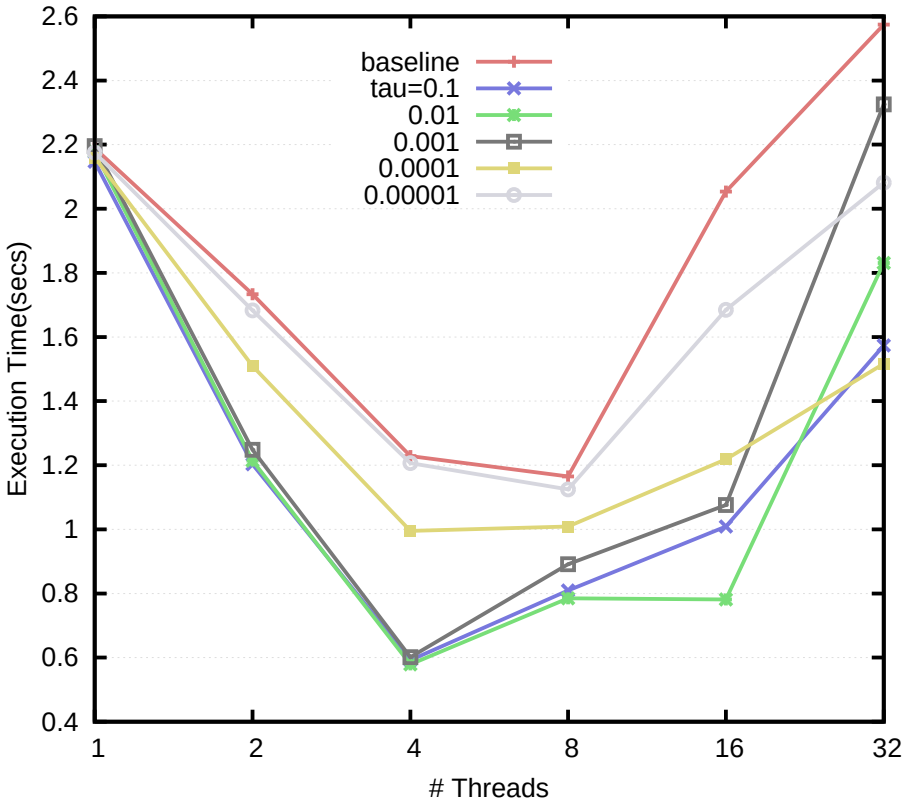
where f_p represents either the velocity or position of the particle p. The velocity $\overrightarrow{V_p}$ calculated for particle p in time-step $t+dt$ depends on the force $\overrightarrow{F_p}$ acting on the particle in time-step $t+dt$, which in turn depends on the distance vectors from particle p to all other particles within a *cutoff-distance* in time-step t. Additionally,

the position $\overrightarrow{P_p}$ of a particle at time $t + dt$ depends on $\overrightarrow{V_p}$ at time t. This sharing of particle positions between threads is the main source of contention for this program. During each time step, new position and velocity vectors are computed for each particle. Depending on the granularity of the time step, the initial velocities and positions, the new vectors can differ from the old ones by very little. If this difference is so small that the old and new position vectors are *approximately-local*, then a consumer thread that consumes the old position vector for a particle need not abort if a new position vector is produced. Therefore a locality threshold can be applied on the shared position vectors to reduce contention among threads by avoiding weak false-conflicts. The performance impact of avoiding strong and weak false conflicts is shown in Figure 24b. The plot shows execution time using relative thresholds and the `REL` operation with τ ranging from 1E-1 to 1E-5. In most cases there is a substantial speedup with a maximum of 2.62x over the baseline.

Several previous works have identified key metrics in measuring the *fidelity* of a particle simulation. These include magnitude of error in linear and angular velocities, error in positions, error in energies etc. For our system the metric most relevant is particle position. In order to calculate error we first compute the Root Mean Square Error (RMSE) in particle positions relative to the outputs produced by the baseline. We then normalize this RMSE with the maximum size of the minimal box that contains all the particles. This is shown by the Normalized RMSE at the end of iteration 1000 in Figure 24a. This figure also shows the rate of growth of error during program execution. We see that this rate of growth is initially roughly linear and starts to reduce towards the end of the program. We also measured the distortion in the outputs produced by distinct baseline runs and we found that as in `kmeans` the RMSE was comparable to this distortion (roughly 40% more). This means that in a relative sense the mean error was comparable to what could be expected out of executions of the baseline program itself.



(a) Growth of error with MV threshold



(b) Execution time with REL threshold

Figure 24: particle

Algorithm 8 `particle`

```
{/* Vector4D: pos, vel, mass, f */}
for time=0; time < NUM_STEPS; time += dt do
  for all {particle "i" ∈ ThisPartition } do
    atomic {
    for all {particle "j" ∈ NeighborWindow} do
      F[i] = computeForces(i, j)
      pos[i] = computePosition(pos[i], dt, vel[i])
    end for
    } endatomic
    vel[i] = computeVelocity(F[i], vel[i], mass[i])
  end for
end for
```

4.7 Related Work

4.7.1 Transaction Nesting

The topic of open nesting in software transactional memory systems has been studied extensively [25, 26]. The main purpose of using open nesting is to separate physical conflicts from semantic conflicts since the programmer usually only cares about the latter. Therefore strict physical serializability is traded for *abstract serializability*. Abstract Nested Transactions [20] allow a programmer to specify operations that are likely to be involved in benign conflicts and which can be executed.

4.7.2 Silent Stores, Value Locality and Reuse

The phenomenon of silent stores has been extensively studied in the computer architecture community [22] and there have been numerous architectural optimizations suggested to exploit the same. Similarly, the phenomenon of *load value locality* has also been studied extensively [11]. Both these concepts basically establish that in many programs, values accessed by loads and stores tend to have a repetitive nature to them. In addition, techniques based on *value prediction* exploit the locality of values loaded in a program to apply optimizations such as cache prefetching. In [21] the authors explore the phenomenon of *frequent values* - values which collectively form

the majority of values in memory at an instant during program execution. In [18], the STM system uses a form of value based conflict detection for improving performance. To our knowledge, this is the only STM system that is explicitly program value-aware. In [19, 16] the authors investigate the detection and bypassing of trivial instructions for improving performance and reducing energy consumption. Frameworks such as memoization [24], function caching [37] and value reuse [41] have been proposed to allow programs to reuse intermediate results by storing results of previously executed FP instructions and matching an instruction to check if it can be bypassed by reusing a previous result.

4.7.3 Relaxed Synchronization and Imprecise Computation

The idea of relaxed consistency systems has been studied in a few contexts. Zucker studied relaxed consistency and synchronization [132] from a memory model and architectural standpoint. In [67], the authors propose a weakly consistent memory ordering model to improve performance. In [28], the authors redefine and extend isolation levels in the ANSI-SQL definitions to permit a range of concurrency control implementations. In [13] the authors propose techniques to provide improved concurrency in database transactions by sacrificing guarantees of full serializability - weak isolation was achieved by reducing the duration for which transactions held read-/write locks. A more recent work [17] work proposes Transaction Collection Classes that use multi-level transactions and open nesting, through which concurrency can be improved by relaxing isolation when full serializability is not required. In [6], the authors propose new programming constructs to improve parallelism by exploiting the semantic commutativity of certain methods invocations.

4.8 Conclusions

A significant body of work exists on characterizing parallel applications in terms of design patterns, memory and cache behaviors, loop-level and task level parallelism

and so on. However a set of significant questions remain largely unexplored: how do shared values in parallel soft-computing applications evolve, is it possible or desirable to synchronize these values imprecisely, what are the accuracy-performance tradeoffs involved? With the rising ubiquity of soft-computing applications these and related questions merit exploration. In this chapter we present the results of our investigation of these questions in the context of three representative workloads.

Conventional optimistic synchronization systems are designed to reason about meta-data of shared data in order to arbitrate conflicts. They consider a store operation as a production of a new value irrespective of the actual value being written. Consequently even if the written value is similar to the original value and the consumer of this value is tolerant of this approximation, it will be found to be in conflict. Hence existing techniques are severely limiting to the parallel performance that these applications can achieve. In this chapter we presented the idea of Approximate Shared Value Locality and a technique to detect its occurrence. We also showed how this technique can be combined with a value based conflict arbitration mechanism to reduce the number of conflicts caused on *approximately local* values. We applied these techniques on a variety of workloads and found that a substantial reduction in abort rate and running time is possible while keeping the error introduced in the results small. In addition the rate of growth of error during execution was small in most cases. In future work we plan to investigate profiling and program analysis techniques that can help the programmer in estimating properties such as rate of growth of the error and the right threshold to use for a particular acceptable level error. It seems likely that these properties cannot be established in a domain-agnostic way or without some involvement from the programmer. Additionally we plan to extend these techniques to be able to reason about more complex program entities like pointers, compound data structures and arrays.

Although we have so far discussed imprecise synchronization in the context of a

software transactional memory system, the broad principles apply to other optimistic synchronization systems like speculative lock elison. Hence another interesting avenue for future work will be to explore and formulate a general framework that is independent of the specific underlying synchronization mechanism.

CHAPTER V

PARALLELIZING A REAL-TIME PHYSICS ENGINE USING SOFTWARE TRANSACTIONAL MEMORY

Applications that simulate the dynamics and kinematics of rigid bodies or physics engines are examples of applications that are known to have significant amount of parallelism but it this parallelism is often difficult to exploit owing to their complexity. Physics engines that support real-time interactive applications such as games are growing rapidly in sophistication both in their feature-set as well as their design. The popular Unreal 3 game engine is known to consist of over 300,000 lines of code and as described in [57], parallelizing parts of it was a challenging endeavour. Traditional approaches to efficient shared data synchronization such as fine-grained locking are often impractical owing to the size and complexity of the application and the large amounts of hierarchical mutable shared state. On the other hand coarse-grained locking has been found to be too inefficient for maintaining the highly interactive nature of these applications. Further, using fine-grained locks in such applications extracts a significant price in terms of programmer productivity - a factor that deeply affects their commercial development cycle.

Researchers have suggested developing parallel programs in this domain using *transactional memory* to manage accesses to shared state [57]. Software or Hardware Transactional memory has been proposed as a relatively programmer-friendly way to achieve atomicity and orchestrate concurrent accesses to shared data. In this model programmers annotate their programs by demarcating atomic sections (using a keyword such as "atomic" in a language-based TM implementation or specific function calls to a library based TM). The programmer also annotates accesses to shared data

within these sections. At run time, these atomic sections are executed speculatively and the TM system continuously keeps track of the set of memory locations each transaction accesses and detects conflicts. This conflict detection step involves checking if a value speculatively read or written has been updated by another concurrent transaction. If so then one of the two speculatively executed transactions is aborted.

Software Transactional Memory systems reduce the burden of writing correct parallel programs by allowing the programmer to focus simply on specifying where atomicity is needed instead of how it is achieved. Further, the benefits of TMs are most apparent when a) the rate of real data sharing conflicts at run time is quite low i.e., most of the concurrent accesses to shared data are disjoint and b) using fine grain locking is difficult either due to the irregularity of the access patterns or the data structures. There has been a substantial amount of interest in hardware and software transactional memory systems recently. However in spite of this recent interest and the significant amount of research most of the studies investigating the use and optimization of these systems have been limited to smaller benchmarks and suites containing small to moderate sized programs [12, 49, 53, 54, 51]. Previous studies [63, 52] have noted the lack of large real-world applications that use transactional memory without which an effective evaluation of the effectiveness of TM systems in realistic settings becomes difficult.

In this section we present our experiences in parallelizing and using transactions in the Open Dynamics Engine (ODE), a single-threaded real-time rigid body physics engine [48]. It consists of roughly 71000 lines of C/C++ code with an additional 3000 lines of code for drawing/rendering. In [52] the authors outline a set of characteristics that are desirable in an application using TM. Briefly they are:

1. Large amounts of potential parallelism: As we show in the Section 5.2, there is a significant amount of data parallelism in the two principal stages in an ODE simulation.

2. Difficult to fine-grain parallelize: ODE exhibits irregular access patterns many structures that can be accessed concurrently.

3. Based on a real-world application: ODE is used in hundreds of open-source and commercial games [48].

4. Several types of transactions: The parallel version of ODE we describe in the rest of this chapter has critical sections that access varying amount of shared data, have sizes that vary widely and the amount of contention between them changes during execution.

We started with the single-threaded implementation of ODE and found that the two longest running stages in a time step could be parallelized effectively. While we found many opportunities for fine-grained parallelization at the level of loops in constraint solvers, we choose to focus on a coarser-grained work offloading in order to amortize the runtime overheads. We then modified this parallel program by annotating critical sections and accesses to shared data with calls to an STM library. Our modifications added roughly 4000 lines of code in the ODE.

The rest of this chapter is organized as follows: Section 5.1 presents an overview of collision detection and dynamics simulation in ODE. Section 5.2 describes the parallelization scheme for ODE and the usage of transactions for atomicity. Section 5.3 briefly discusses a few issues pertaining to the parallelization. Section 5.4 presents our experimental evaluation of the application and Section 5.5 concludes the chapter.

5.1 ODE Overview

At a high level ODE consists of two main components: a collision detection engine and a dynamics simulation engine. Any simulation involving multiple bodies typically uses both these engines. The sequence of events in a typical time step is shown in Algorithm 9. The goal is typically to simulate the movement of one or more bodies in

Algorithm 9 Overview of a time step in ODE

```
 1: Create world; add bodies
 2: Add joints; set parameters
 3: Create collision geometry objects
 4: Create joint group for contact points
 5: // Simloop
 6: while (!pause && time < MAX_TIME) do
 7:   Detect collisions; create joints
 8:   Step world
 9:   Clear joint group
10:   time++
11: end while
```

a *world*. Before simulation begins the world and the bodies in it are created and any initial joints are attached. A *contact group* is created for storing the contact joints produced during each collision. During each time step in the simulation loop in line 6, *collision detection* is first carried out which creates contact points/joints which are used in "stepping" or *dynamics simulation* for each body in the world (line 8). After this step all the contact joints are removed from the contact group and the simulation proceeds to the next time step.

5.1.1 Collision Detection

The collision detection (CD) engine is responsible for finding which bodies in the simulation touch each other and computing the *contact points* for them given the shape and the current orientation of each body in the scene. A simple algorithm would simply test whether each of the "n" bodies collides with any other body in the scene but for large scenes this $O(n^2)$ algorithm does not scale. One solution to this problem is to divide the scene into a number of *spaces* and assign each body to a space. Additionally, the spaces may be hierarchical - a space may contain other spaces. Now, collision detection proceeds in two phases called *broadphase* and *narrowphase* which are as follows:

1. **Broadphase**: In this phase each space $S_1(\in S)$ is tested for collision with each of the other spaces. If S_1 is found to be potentially colliding with space $S_2 \in S$ then S_1 is tested for collision with each of the spaces or bodies inside S_2.

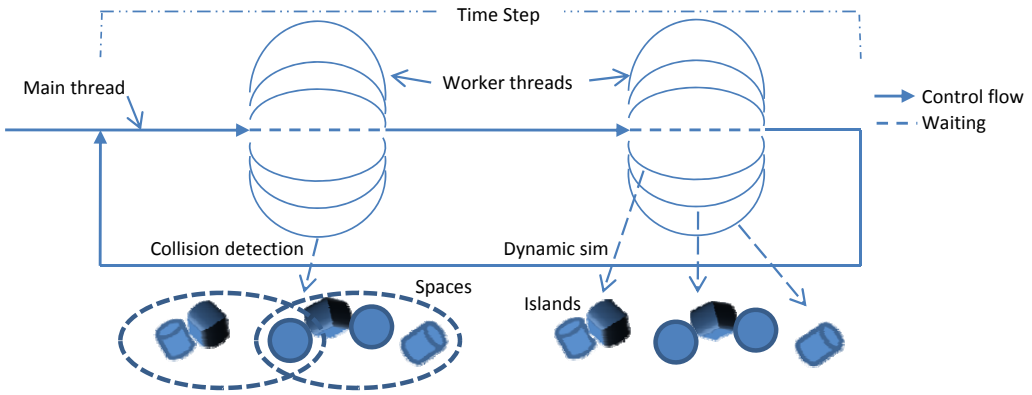
2. **Narrowphase**: In this phase individual bodies that have found to be potentially colliding in the broadphase are tested to check if they are actually colliding.

This approach is similar to the hierarchical bounding box approach used for fast ray tracing and many other problems. If a pair of bodies are found to be colliding the collision detection algorithm finds the points where these bodies touch each other. Each of these contact points specifies a position in space, a surface normal vector and a penetration depth. The contact points are then used to create a joint between these two bodies which imposes constraints on how the bodies may move with respect to each other. In addition to links to the bodies each of these *contact joints* connect, they also have attributes like surface friction and softness which are used in simulating motion in the next step.

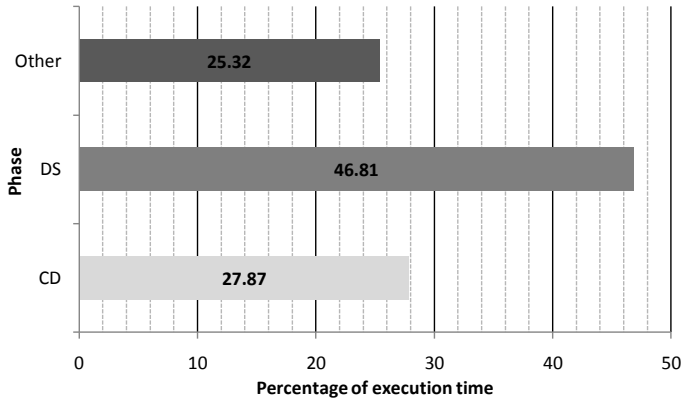
By the end of the collision detection step all the contact points in the scene have been identified and the appropriate joints between bodies made. In the dynamics simulation step below, the new positions and orientations of all the bodies in the scene are computed.

5.1.2 Dynamics Simulation

The joint information computed in the CD step above represents constraints on the movement of the bodies in the scene (for example due to another body in way or due to a hinge). The Dynamics Simulation (DS) engine takes this joint information and the force vectors and computes the new orientation and position for all the active bodies in the scene. It does this by solving a Linear Complementarity Problem (LCP)



(a) Overview of parallel ODE



(b) Distribution of execution time among phases in single-threaded execution

Figure 25: ODE overview

using a successive over-relaxation (SOR) form of the Gauss-Seidel method. The main output produced in the DS stage are the linear and angular velocities of each body in the scene. These velocities are then used to update the position and orientation of the bodies.

5.2 Parallel Transactional ODE

The broad approach to parallelizing ODE is illustrated in Figure 25a. At a high-level parallelism is achieved by offloading coarse-grained tasks in the CD and DS stages on the main thread onto concurrent worker threads that use transactions to synchronize shared data accesses.

5.2.1 Global Thread Pool

In order to avoid the overheads of creating and destroying threads, before the simulation begins the main thread creates a global thread pool consisting of t POSIX threads that are initialized to be in a *conditional wait* state. Additionally the pool contains a t-wide status vector that describes each thread's status, a set CM of t mutexes and a set CV of t condition variables. During the course of the simulation the main thread offloads work to a worker thread by scanning the pool for an idle thread, marshalling the arguments and setting the condition variable for the thread to start execution.

5.2.2 Parallel Collision Detection using Spatial Decomposition

Detecting collisions between bodies in the world is inherently parallel and indeed the naive $O(n^2)$ algorithm described above can be parallelized by simply performing collision detection for each pair of bodies in a separate thread. However a better scheme would involve a more coarse-grained distribution of work in which a space or a pair of spaces in the world is handled by a separate thread. Before the parallel CD stage starts each of the bodies in the world is assigned to a space S_i. Let S represent the set of spaces in the world i.e., $S = \bigcup_i S_i$. Detecting collisions among bodies contained in the same space can be done independently of (and in parallel with) other spaces. Additionally, detecting collisions between each distinct pair of spaces can be done in parallel. The broadphase stage of parallel CD proceeds as follows.

1. The main thread picks an unprocessed pair of spaces S_1 and S_2 and signals an idle thread $t_{1,2}$ in the thread pool to perform collision detection on them. Additionally the main thread signals idle threads t_1 and t_2 to perform collision detection on bodies contained withing S_1 and S_2 respectively.

2. Thread $t_{1,2}$ first checks if spaces S_1 and S_2 can potentially be touching. It does this by checking if there is an overlap between their axis aligned bounding boxes (AABBs). As described above, the AABB for a space informally is simply the smallest axis aligned box that can completely contain all the bodies in that space. If there is overlap between the AABBs of the two spaces then $t_{1,2}$ has to check if there exist bodies b_1 and b_2 such that $b_1 \in S_1$, $b_2 \in S_2$ and the AABBs of b_1 and b_2 overlap. If they do, b_1 and b_2 are potentially colliding and the narrowphase later on checks if they are actually colliding. After this step thread $t_{1,2}$ marks the space pair (1, 2) as processed.

3. Thread t_1 finds bodies in S_1 that are potentially colliding. This is done again by analyzing the AABBs of bodies in S_1. Thread t_2 does the same for bodies in S_2. Spaces S_1 and S_2 are then marked as processed by their respective threads.

4. All the potentially colliding bodies found above are checked to find actual collisions in the narrowphase. If a pair of bodies do actually collide the appropriate thread computes contact points for the collision (using the positions and orientations of the bodies). These contact points are used by the thread to create contact joints between the pair of bodies.

This approach to assigning collision spaces to threads makes $(\binom{n}{2} + n)$ thread offloads where n is the number of spaces. An alternate approach is to assign a single thread t_i to each space S_i. This thread computes the collisions for objects within S_i and then performs broadphase and narrowphase collision checking between S_i and all S_j such that $i < j \leq n$. This approach activates only n threads but is likely to be more efficient than the former only if the spaces are *well balanced*. That is all

the spaces at each level in the containment-hierarchy contain approximately the same number of subspaces or bodies. Consider a deep space hierarchy with space S_{root} as the root space that contains all other spaces S_i and bodies. In the alternate approach the thread t_{root} has to process collisions between S_{root} and all other spaces/bodies. By definition, S_{root} would collide with every other contained body or space. Thus in general this approach would result in a schedule where threads processing spaces that are high-up in the hierarchy are heavily loaded while threads assigned to spaces that are lower are lightly loaded. However in the former approach, each space-space pair can be processed in parallel - each pair $\{S_{root}, S_j\}$ for $1 < j \leq n$ can be processed in parallel thereby reducing the overall imbalance.

Shared data

Although the collision detection stage described above is quite parallel the participating threads make concurrent accesses to several shared data structures that must be synchronized. The important data structures that are accessed concurrently are the Global Memory buffer that is used to satifsy allocation requests, the joint, contacts and body lists and attributes pertaining to the state of the world and its parameters including the number of active bodies and joints.

We use an STM library to orchestrate calls to these shared data. STM enables efficient disjoint access parallelism - two concurrent threads that do not access the same memory word can execute in parallel. This is in contrast to using more pessimistic coarse-grained locking in which a thread that *could* access/modify shared data (being accessed by some other thread) has to wait to acquire the appropriate lock regardless of whether an actual access takes place or not. The STM library we used is based on the well-known TL2 system described in [1]. In other works such as [63] the authors used an automated compiler-based STM system in which the programmer simply annotates atomic sections and the compiler automatically annotates accesses

occurring inside them with calls to the TM runtime. Instead we used the TL2 library based system which means the programmer has to manually identify atomic sections and accesses occurring within them. This choice is because of two reasons. Firstly the TL2 STM has been shown to have lower overheads than other comparable STM systems in several studies [1]. This is especially important since we are using it in the context of a real-time interactive application. Secondly using a library STM offers better flexibility and we are in some cases able to reduce TM overheads by using domain knowledge to elide TM tracking of specific shared data.

5.2.3 Parallel Island Processing

Island Formation

After the joints in the world have been determined in the CD step the next stage is dynamics simulation or simulating the motion of the bodies under the constraints specified by their shapes and the joints found. This uses the SOR-LCP formulation mentioned above and finding solutions to this problem involves several nested loops that are compute-intensive. However, parallelizing these loops with the work-loading model would result in a very fine-grained parallel system (which is unlikely to scale well [56] and the overheads of synchronization and thread control would likely eliminate any speedups gained. Therefore we choose a more coarse-grained approach in which several connected bodies are processed independently and in parallel with other bodies. All the bodies in the world are assigned to "islands". An island is simply a group of bodies in which each body is connected to one or more bodies in the same island through one or more joints. These islands therefore represent sets of connected bodies that can be processed separately since simulating a body (with some number of joints) does not require accesses to bodies in other islands. In parallel dynamics simulation the main thread first forms islands. The algorithm iterates over all the bodies in the world adding bodies to islands if they haven not already been added. A

body is said to be *tagged* when it has been added to some island. Given a body b, the algorithm first finds the untagged neighbors of b and adds adds them to a stack. The algorithm then pops and examines each body in this stack, adding their untagged neighbors. The joints between all these neighbors are collected in a joint list. When the stack is empty, the joint and body lists represent an island of connected bodies that can be processed. The main thread then moves on to the next untagged body in the world in the outermost loop.

Island Processing

While island formation is sequential, processing the bodies in each island can be performed independently of other islands. Immediately after an island is formed, the main thread uses heuristics to check whether the island is suitable to be offloaded to a worker thread. If so, the main thread marshals pointers to body and joint lists for that island, finds an idle thread in the global thread pool and signals it to start processing that island. The main thread then resumes with finding the next island. If the island formed is deemed to be not suitable for offloading, the main thread can process that island itself before continuing with further island formation. A variety of heuristics can be used to decide whether a particular island should be processed in a worker thread or if it should be processed in the main thread. Our system uses a threshold on the number of bodies and number of joints in the island. Because of the overhead of offloading computation to worker threads, if there are very few bodies or joints in the islands then it may be more efficient to process them in the main thread instead. Additionally, if an island is found to have fewer bodies than needed to offload processing to a worker thread, the main thread checks whether the next island in combination with the previous one meets the threshold. If so both these islands are offloaded together to a single worker thread. The main thread chooses and signals a thread from the global thread pool to start island processing.

The worker thread uses the body and joint lists and the force vectors to set up a system of equations representing the constraints on the set of bodies and finds. We refer the reader to [48] for details of the constraint solver that is used for finding solutions. The island processing step finishes after computing new values for linear and angular velocity, position and orientation quaternion for each body in the island and atomically updating body with these values.

5.2.4 Phase Separation

During body simulation in ODE, all the contact joints are typically computed first before dynamics simulation can start since the latter needs these joints to be able to solve the constraint satisfaction problem. In the sequential case this was guaranteed since the dynamics simulation is always preceded by collision detection in each time step. However in the parallel case, the main thread can simply offload the collision detection to worker threads and enter the dynamics simulation step while some of the worker threads are still computing the joints. Therefore there needs to be a thread barrier between the collision detection and dynamics simulation in simulating each time step. The control flow for the main thread is very different from that of the worker threads in our parallelization scheme. Therefore instead of a normal thread barrier that is released when all threads reach a certain program point, in our scheme we use a thread *join point* in the main thread. A *join point* is simply a program point at which the main thread waits for all the active worker threads to finish executing. When the main thread enters the join point, it repeatedly polls the *status* vector and yields its processor if there is at least one worker thread performing collision detection. Note that no lock acquisition is necessary for this polling as the worker thread only ever writes one type of value into its slot in the status vector - the value representing its *IDLE* state. After all worker threads have finished collision detection and have entered the *IDLE* state, the join point is met and the main thread is released.

Although it limits parallelism, this join is necessary due to the producer-consumer relation between the stages for joints - the island formation algorithm requires contact joints for all bodies in the world to have been computed.

After island processing has generated new positions and orientations for all the bodies in the world, these new values are used in the collision detection step in the next stage. But after the main thread offloads island processing to worker threads, it could enter the collision detection stage in the next time step while the new body attributes are being computed. This could result in the collision detection stage reading stale position/orientation values for some bodies - the bodies which island processing has not yet updated. Therefore in addition to the dependence between the collision and dynamics simulation steps within a time step there is also a dependence between the dynamics simulation in one time step and the collision detection in the next. We therefore enforce a join point at the end of each time step to make sure that all bodies have been updated. This join point is implemented like the one described above - the main thread simply polls the status vector until all the island processing worker threads have finished.

To see why this join point is needed consider the case of a worker thread with transaction Tx_1 updating the position quaternion R_b of a body b during island processing in time step n. Assume the main thread is allowed to enter the next time step where it offloads collision detection to a worker thread and transaction Tx_1 is reading R_b. If Tx_1 commits after Tx_2 starts but before it finishes then Tx_2 is aborted when the conflict for R_b is detected and the join point would *not* have been necessary. However if Tx_2 commits before Tx_1 does, then Tx_1 is aborted and retried. Thus Tx_1 eventually produces the new value for R_b but Tx_2 ends up using the older value and this phenomenon can adversely affect simulation integrity. Now lets say add a "*last_updated*" field to each body which is updated in Tx_1. So if Tx_2 finds this field for b to be n then Tx_1 is guaranteed to have committed and Tx_2 can read

the latest R_b. However if this value is $n-1$ then Tx_2 can be forced to abort to until Tx_1 commit. It may therefore be possible to eliminate the join point at the end of each time step by forcing transactions reading stale values in the next time step to abort. This could potentially allow more parallelism by allowing the threads with transactions that only read already updated bodies to proceed instead of waiting for the other threads.

5.2.5 Feedback between phases

A critical factor influencing the amount of effective parallelism achieved during the CD phase is the assignment of bodies to spaces. Spatial (in the geometric sense) assignment methods are popularly used in many dynamics simulation algorithms. In such methods, objects that are geometrically proximal to each other are assigned to the same space in the containment hierarchy. An important concern with this approach is that the scene being modelled may evolve to a state where most of the objects are contained in one or a few spaces. This may in turn result in the *thread imbalance* problem discussed in Section 5.2.2. To address this such methods usually propose a space reassignment step that is invoked occasionally and reassigns objects such that the threads are once again balanced. We use a novel method to perform space assignment that reduces imbalance. Our method is based in the observation that the DS phase in a timestep already computes entities (islands) of geometrically close bodies - in fact the bodies in each of these islands are touching each other! After the dynamics simulation step, the bodies in these islands have been moved so they may not be touching anymore. However if the simulation timestep is small then in the CD phase in next iteration these bodies are either still touching each other or are close to each other. Hence the CD phase bootstraps spaces with clusters of such islands before performing broadphase checks on these spaces with the result that there are fewer narrowphase checks to be performed on the contained bodies.

5.3 Issues

In this section we will discuss a few issues pertaining to using transactions for synchronization in parallel ODE.

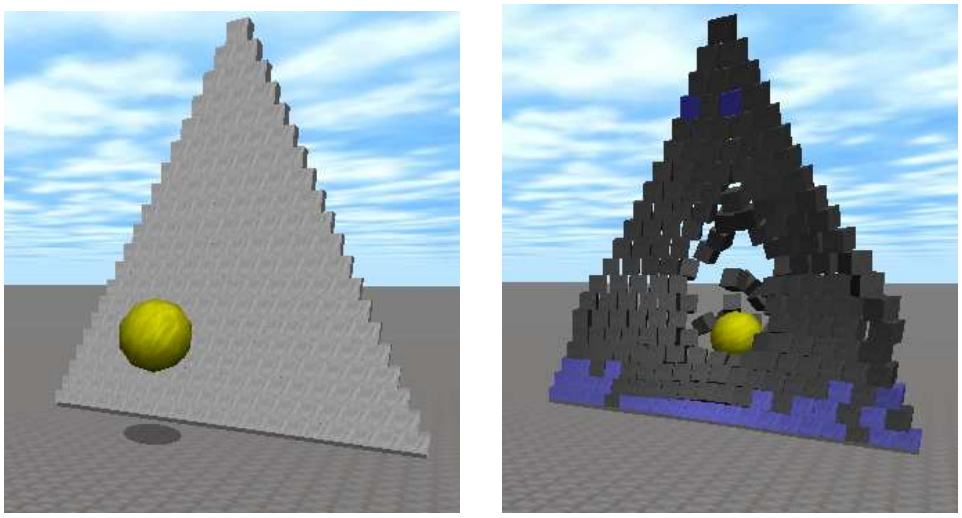
5.3.1 Conditional Synchronization

Our implementation of parallel ODE makes extensive use of conditional synchronization for signalling between threads. Indeed constructs such as `pthread_cond_wait` and

`pthread_cond_signal` enable efficient waiting, signalling and other communication between threads. However these constructs require the communicating threads to acquire/release locks during doing so. Moreover there is no direct way to transform these critical sections into transactional atomic sections. Consider the case of a worker thread t_w waiting for the main thread t_M to offload work. The thread t_w first acquires a lock on the *waiting mutex* l and calls `pthread_cond_wait(..,`l). This call atomically unlocks the mutex and starts the conditional wait. To signal thread t_w to start execution, the thread t_M in turn acquires a lock on l, calls `pthread_cond_signal()` and releases the lock on l. If the critical section protected by the lock acquisition/release in t_M were to be transformed into an atomic section using transactions, then if there is a conflict in the transaction in t_M the transaction cannot roll back since the signal has been set and it is irrevocable. Most STM systems including the TL2 system we used and the compiler-based STM in [55] do no provide transactional methods for conditional synchronization and signalling. Consequently our implementation uses traditional mutex based methods for conditional synchronization.

5.3.2 Memory management and application controlled alloc/de-alloc.

Dynamic memory allocation is another important programmatic concern for STMs. Most STM systems provide methods for allocating and deallocating memory efficiently from within transactions. Additionally they often implement a large memory buffer from which allocations are made and of course memory that is allocated in a failed transaction is restored back to the buffer. Many of the important classes of objects in ODE are allocated dynamically on the heap. This includes bodies, joints, joint lists, and other shared data. However, ODE implements its own memory allocation/deallocation algorithms that purport to improve locality and to allow objects to be be efficiently garbage collected in addition to implementing its own large stack-shaped buffer from which allocation requests are met. Requests for memory allocations are made using the `ODE_Alloc()` which simply returns a pointer to the first location in memory that has not previously been allocated. If concurrent transactions in two different threads call `ODE_Alloc` at the same time, both may receive the address of the same location in memory. And as with all transactional writes to shared data, the modifications they make to this newly allocated memory region will be buffered in their respective private write-buffers. Suppose one of them finishes and commits successfully. At this point its modifications to the heap will actually be written to memory. When a conflict is detected when the second transaction tries to commit it will be aborted. As the TM runtime rolls this transaction back, the memory allocated within it will be freed thereby freeing memory that the first transaction is using. Therefore the memory allocation/deallocation library should be modified to be aware of the revocable nature of allocations. For programs that may make use of such routines from one or more of several external libraries this is a significant problem.



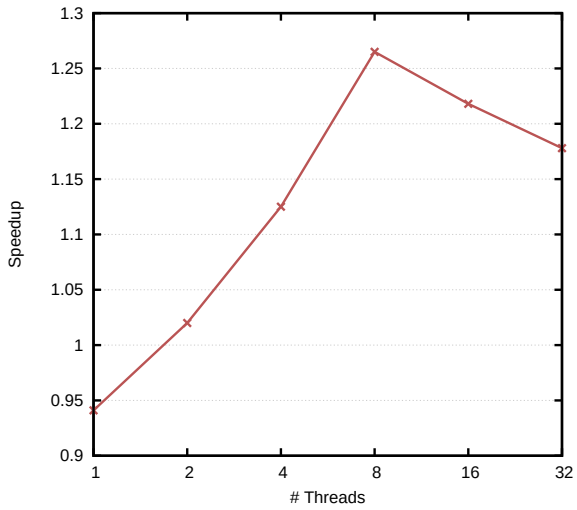
(a) (b)

Figure 26: Scene used in evaluating parallel ODE

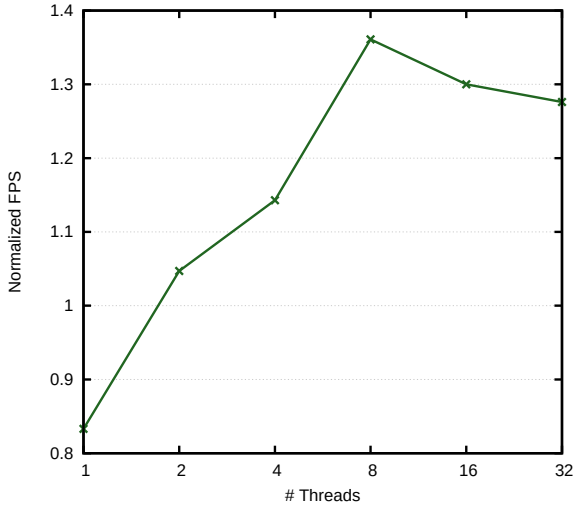
5.4 Experimental Evaluation

We used the parallel ODE library in to drive an application simulating a scene with approximately 200 colliding rigid bodies (a modified version of the `crash` program in the ODE distribution). The maximum number of worker threads in the global thread pool was varied from t = 1 to 32 in powers of 2. The number of threads in the results below therefore represents the maximum number of worker threads available to for offloading and the maximum number of active threads at any instant is (t+1) including the main thread.

We used the TL2 (v0.9.6) STM [1] API and library to provide support for transactions in the ODE library as well as in the driver application program. This version of TL2 is a *word-based write-buffering* STM that uses *lazy version checking* for detecting conflicts and *commit-time locking*. All experiments were carried out on a machine with an Intel Xeon dual processor with two cores per processor and with hyperthreading turned on on all cores (for a total of 8 thread contexts). This in our opinion represents an average platform that may be used to run interactive simulations in ODE. Machines with higher core counts such as (8 or 16) are less common



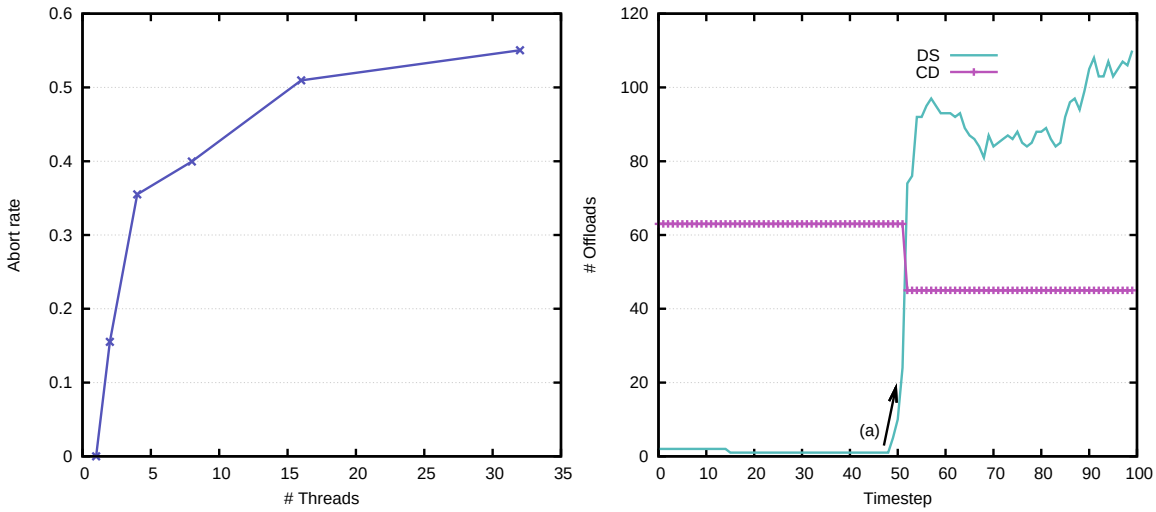
(a) Performance relative to single-threaded ODE



(b) Effect on Frames per Second

Figure 27: Scalability

(although they are available) and servers with core counts of 32 and more are less frequently used in running these predominantly desktop oriented simulations. Each core on this machine had a private 32K L1-D cache, 32K L1-I cache, a shared 256KB L2 per processor and a shared 8MB L3 cache and the machine was equipped with 6GB of physical memory. Each thread in our experiments was bound to exactly one core. We compiled all libraries and the driver application with g++-4.3.3 using the default flags and all experiments were run on Ubuntu Linux 2.6.28. All running times were gathered using the `gettimeofday()` call.



(a) Normalized abort rate

(b) Number of offloads to worker threads per timestep (or frame)

Figure 28: Aborts and Offloads

5.4.1 Execution time

The graph in Figure 27a shows the improvement in execution time as speedup over the single-threaded execution time. The X-axis is the *maximum number of threads available for offloading*. The speedup scales until 8 threads at which point it is roughly 1.27x. At 16 and 32 threads it drops to roughly 1.22x and 1.18x approximately. This means that the heuristics may be too aggressive in offloading work when idle threads are available. This hurts performance since there may not be enough work for a worker thread (not enough joints or bodies in island processing for example) to justify the overhead of offloading. Moreover, at 16 and 32 threads each core is utilized by 2 and 4 threads respectively which means increased contention may also be responsible.

5.4.2 Frame rate

Figure 27b shows the number of frames processed per second (FPS) against the number of threads in the thread pool. In our experiments each time step corresponds to one frame. The frame rate scales in a trend similar to that of execution time speedup. The improvement in frame rate peaks at 1.36x and drops to 1.27x for 32

threads. At more than 8 threads more than one thread is mapped to a processor and contention for shared data also increases reducing the per frame completion time.

5.4.3 Abort rate

The abort rate for different number of threads is shown in Figure 28a. The abort rate is defined as the ratio of the number of aborts to the total number of transactions started. Therefore if a, c represent number of aborts and commits, the abort ratio is given by $a/(a+c)$. The abort ratio increases steeply up to 4 threads and continues to rise beyond. The average amount of contention between threads increases as the number of threads increases and the amount of shared data being accessed by these threads remains the same. The abort rate does not rise as significantly going from 16 to 32 threads. This is because the average number of *concurrent* threads does not necessarily rise proportionally to the number of threads in the thread pool and therefore the number of aborts increase less steeply.

Table 5: Read/Write set sizes

	Reads (bytes)				Writes(bytes)			
Threads	Min	Max	Avg	Total	Min	Max	Avg	Total
1	4	112	112	3094332	4	96	48	1325062
2	4	224	211	5886756	0	192	90	2520386
4	4	2536	596	16620560	0	2036	240	6791206
8	4	2868	1300	36245344	0	2328	530	14775982
16	4	3552	1393	38823380	0	2936	570	15868776
32	4	5184	1504	41912768	0	4196	614	17133684

5.4.4 Thread utilization

In contrast to parallelization techniques that purely depend on static decomposition of work, in the scheme for parallel dynamics simulation (DS) described above, only the maximum number of threads in the thread pool is fixed and heuristics are used

to dynamically gauge whether to offload island(s) processing to worker threads. The amount of parallelism in the collision detection (CD) stage however remains relatively uniform. The plot in Figure 28b shows the average number of computation offloads occurring in each time-step (or frame) when there are a maximum of 32 threads in the global thread pool. Specifically, the plot shows the number of offloads to worker threads for the first 100 frames of simulation for the scene shown in Figure 26. The number of offloads in the CD stage remain stable and in this stage, a worker thread can be invoked on average roughly 2 times until the point in the simulation noted as *(a)* in the plot. Also, the number of offloads in the DS stage remains low and is also stable until point *(a)*. This is the time step where the stack of bodies in Figure 26 begins to disintegrate as shown in Figure 26(b). While in earlier time steps there was only one island to process, after point *a* there are many smaller islands and therefore there is more parallelism. This is reflected in Figure 28b by the sharp increase in number of offloads in the DS stage after point *(a)*. As mentioned above, the heuristics we used have a relatively low threshold on island count for offloading the work of processing an island to a worker thread. This results in the main thread aggressively offloading work which explains the high number of DS offloads after point (a). The number of offloads in the CD stage remain relatively stable since there the data distribution is based on abstract spaces and not physical artifacts such as joints and islands. Additionally, after point *(a)* the number of offloads in the CD stage are reduced due to contention with the DS stage for worker threads.

5.4.5 Transaction Read/Write Sets

There are three main types of transactions during execution. The first is the transaction to add a contact joint to the system for a pair of colliding bodies. The second transaction executed during island processing for atomically updating a body's attributes. The third type consists of short transactions to access various shared values

such as the number of joints. Table 5 summarizes the characteristics of the read/write sets of all the transactions executed. The average read set sizes are significantly larger than the sizes of the write sets in all cases. This is in line with the average mix of read/write operations in many other transactional programs. Many of the transactions in parallel ODE perform several reads before performing their first write. One commonly occurring transaction for example is atomic insertion into a sorted object list. Here the list is traversed and each element examined to find the right position for insertion before pointer values for the neighboring list elements are updated. The average read and write set sizes remain relatively small for most transactions which shows that hardware transactional memory implementations may also be able to support parallel ODE.

5.4.6 Scalability Optimizations

Based on the results of the experiments described above, the following observations can be made pertaining to improving scalablity.

1. DS phase offloading: The work offloading algorithm in the island processing phase may be too aggressive in our experimental system. This stems partly from the static threshold used to decide whether processing for a particular island is to be offloaded, inlined or whether it should be combined with another island and then offloaded. The size of the islands changes substantially over the course of the simulation (for example, the one shown in Figure 26a), which results in the threshold becoming too low at several points. A low threshold results in aggressive offloading which in turn results in poor scalability. The processing step for a single island cannot be offloaded to more than one thread in our system. This is because the forces and torques acting on a body are determined by the joints connecting the body to its neighboring bodies and if these bodies were being processed by two separate threads the system of

constraints imposed by these joints would have to be communicated between them which we believe would increase the level of synchronization drastically. During the early timesteps of simulating the scene shown above, there are only two islands with one of them containing all the bodies in the world and this large island is then offloaded to a single thread. This restriction therefore has the effect of severely serializing island processing until more islands are formed as a result of collisions.

2. Performance of Locks: Coarse-grained locking can be used instead of transactions to protect accesses to shared state and we believe that the performance in both cases would be comparable. Fine-grained locking would be harder to implement given the diversity of both the data structures and the accesses to them. Nevertheless we are in the process of implementing our parallel ODE system with support both coarse-grained and fine-grained locking.

3. Speculative island formation: The algorithm for discovering islands discussed earlier is sequential - the main thread discovers an island and offloads (or inlines) it before proceeding to discover the next island. This substantially limits the amount of effective parallelism especially for very large scenes. An algorithm for speculatively discovering islands in parallel and processing them in the worker threads after the speculation has been verified would improve parallel performance greatly (in spite of the additional synchronization costs which are relatively small). Briefly, in this algorithm worker threads speculate on a "seed" body for an island and then "grow" the island. This seed body is picked from a cache of likely candidates built during the island discovery phase in the previous timestep. The worker threads then attempt to verify if the island is valid and was previously undiscovered and if so, continue to the island processing step.

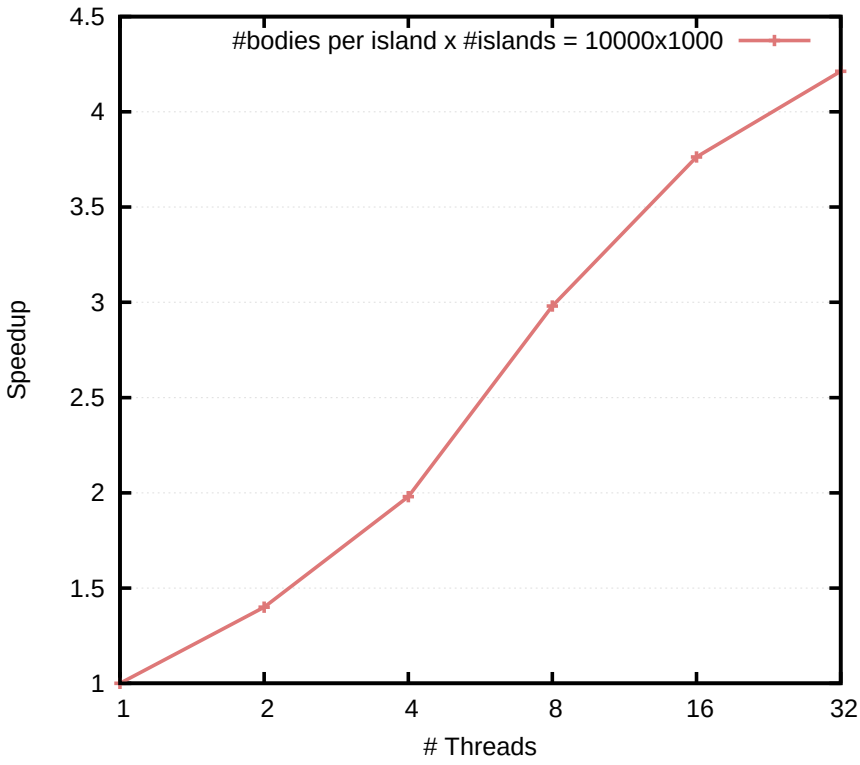


Figure 29: Speedup in speculative parallel island discovery relative to the single-threaded algorithm. The speculative version is *conflict-free* and *synchronization-free* in this case

The island discovery algorithm is a variant of Tarjan's algorithm for finding connected components in undirected graphs where each body is a node in a graph and the the edges between nodes represent physical joints connecting bodies. The plot in Figure 29 shows the speedup in island discovery for a world consisting of 1000 islands and 10000 bodies per island (a total of 10 million bodies). The speedup for n threads is measured as the ratio s/t_n where s is the time taken for one step of island discovery by the sequential algorithm on this scene and t_n is the time taken when $(n-1)$ speculative tasks are launched in addition to the non-speculative main thread. The plot shows that this method of parallelization achieves substantial speedups and more importantly scales well for n.

5.5 Conclusion

In this chapter we presented a parallel transactional physics engine for rigid body simulation based on the popular Open Dynamics Engine (ODE). We were able to parallelize the two principal components of ODE - the collision detection engine and the dynamics simulation engine to make use of worker threads from a global thread pool for executing work offloaded from the main thread. We used a software transactional memory for orchestrating concurrent accesses to all shared data. Our approach of coarse-grained parallelization was not only relatively programmer friendly but also helped amortize the cost of the work-offloading. The parallel version of ODE showed speedups of up to 1.27x (for 8 threads) compared to the sequential version. As a continuation of this work we plan to investigate better cost heuristics for making offloading decisions and to investigate techniques for incorporating domain knowledge in optimizing memory transactions in addition to comparing the performance of the transactional implementation with that of versions that use fine-grained and coarse-grained locking.

CHAPTER VI

A RELAXED-CONSISTENCY TRANSACTION MODEL

The consistency property in database and memory transactions guarantees that all the shared variables read in a transaction are consistent as according to some serializable schedule of all the transactions in the system. However in some programs such consistency may be required only on a specific set of variables. That is, some sets of variables are required to be consistent and the others variables accessed in the transaction are not. Consider the example of a game engine that models a set of movable objects (players, weapons, vehicles, projectiles, particles, arbitrary objects etc). Each of these game objects is represented by a program object that has among others, three mutable fields representing x,y,z positions of the object at an instant. The game object can be subject to many factors that change its position - game play factors like user input, movement due to being attached to other bodies in a joint, physical forces like collision with another body and so on. The program object representing this game object is shared among all the modules implementing those forces. This program object is (or atleast the fields in that object are) thus potentially touched by a very large number of writers. It is also accessed by a large number of readers. For example, the rendering engine reads the position fields in order to perform the visibility test and to draw the object into the graphics frame-buffer. Other readers of these fields could include physics modules that perform collision detection, and scripts that trigger events based on the players proximity. However the position fields need not be accurate on every frame and all the readers do not need the most up-to-date values to execute correctly. For example, reading accurate position values in collision detection may be more important than in triggering events like special

effects. Additionally, the modifications made by all writers are not equally important and some modifications can be safely ignored. For example, minor modifications to a moving particle's position due to wind or gravity can be safely ignored from frame to frame. Such semantics are at best clumsy or at worst not possible to express with current TM programming models.

In this section we describe a relaxed consistency model for STM that enables a programmer to express these parallel programming and synchronization idioms.

6.0.1 RSTM

Parallel programs such as threaded game engines, interactive physics simulation and animation programs are very good candidates for using STM [125]. They have the following important features that are interesting to us.

1. Large amount of shared state - threads spend a significant portion of their execution time inside critical sections. Having a lot of shared state implies that a standard STM will suffer from large number of roll-backs.

2. High performance (frame-rates, number of game objects) and providing a smooth user perception is absolutely critical. Current STM implementations are known to suffer from large performance overheads [126].

3. There are large existing C/C++ game code-bases that use lock-programing. These code-bases are proving hard to scale to quad-core architectures.

4. The actual fidelity to real-world physics is not important so long as the user-experience is smooth and appears realistic. Therefore, not all computation has to be completely accurate.

5. Game applications are the biggest application domain till now to make use of multicores. A high-performance parallel programming model that maintains

ease of use (verification, productivity) while scaling well with the number of cores, would be highly desirable.

Consider this example that is representative of scenarios in many games. There are a set of movable objects (players, weapons, vehicles, projectiles, particles, arbitrary objects etc). Each of these game objects is represented by a program object that has among others, three mutable fields representing x,y,z positions of the object at an instant. The game object can be subject to many factors that change its position - game- play factors like user input, movement due to being in contact with other bodies (a vehicle for example), physical factors like wind, gravity, collision with a projectile and so on. The program object representing this game object is shared among all the modules implementing those factors. This program object (or atleast the fields in that object) is thus potentially touched by a very large number of writers. It is also accessed by a large number of readers. For example, the rendering engine reads the position fields in order to perform the visibility test and to draw the object into the graphics frame-buffer. Other readers of these fields could include physics modules that perform collision detection, and gameplay modules that trigger events based on the players proximity. The following observations hold for the described game scenario:

1. The position fields need not be accurate on every frame. Many times, stale values will suffice. Regular STMs do not take advantage of this. All readers do not need the most up-to-date values to execute correctly. For example, reading accurate position values in collision detection may be more important than in triggering events like special effects. RSTM group consistency semantics allow optimizing for this scenario where deemed desirable and safe by the programmer.

2. The modifications made by all writers are not equally important - some modifications can be safely ignored. For example, minor modifications to a moving

particle's position due to wind or gravity can be safely ignored from frame to frame. RSTM incorporates this by allowing a prioritization of writes to specific variables between concurrent transactions.

6.0.1.1 Constraints

While games fit our programming model well, they also impose certain constraints on the implementation of the STM. The most important constraint is that games are written in C/C++ because of the low-level tweaking that this language allows. This imposes that our STM implementation works in C/C++. The most important consequence of this constraint is that atomicity book-keeping cannot be done at an object level as pointers allow access to virtually any point in memory. An object could be modified without going through an identifiable language construct. We thus propose a solution with a byte-level book-keeping with optimizations to limit the amount of book-keeping required.

6.0.2 Contributions

This work makes the following contributions:

- *Relaxed STM* is a new STM model that allows a relaxation of the atomicity constraint for traditional STM.
- *C-language RSTM extension* allows the programmer to directly specify transactions and relaxation constraints for each transaction. We have implemented a source-to-source translator for our language extensions, as well as a purely C API based implementation.
- *Zone based memory management* allows efficient management of book-keeping at varying granularity levels that are dynamically determined.

The rest of this chapter is organized as follows. Section 6.2 introduces RSTM and describes our language extension. Section 6.3 focuses on implementation. Section 6.4

presents our experimental result and section 6.6 concludes the chapter.

6.1 Relaxed consistency STM

The relaxed consistency STM model (RSTM) extends the basic atomicity semantics of STM. The extended semantics allow the programmer to **i)** specify more precise constraints in order to reduce unnecessary conflicts between concurrent transactions, and **ii)** allow concurrent transactions that take a long time to complete to better co-ordinate their execution. This allows the semantics of a regular STM to be weakened in a precise manner by the programmer using additional knowledge (where available) about which other transactions may access specific shared variables, and about the program semantics of specific shared variables. The atomicity semantics of regular STM apply to all transactions and shared data about which the programmer cannot make suitable assertions. The two primary mechanisms for relaxed semantics are described in the following subsections.

6.1.1 Conflict Reduction between Concurrent Transactions

Problem Conflict-sets can be large in regular STMs, leading to excessive rollbacks in concurrent transactions. This problem scales poorly with increasing numbers of concurrent threads.

Opportunity Game Programmers approximate the simulation of the game world. They are very willing to trade-off the sequential consistency of updates to shared data in order to gain performance, but only to a controlled degree and only under specific execution scenarios. The execution scenarios typically depend on which specific types of transactions are interacting, and what shared data they are accessing.

Our Solution Programmers can assign labels to transactions, and identify groups of shared variables in a transaction to which relaxed semantics should be applied. The relaxed semantics for a group of variables are defined in terms of how other transactions (identified with labels) are allowed to have accessed/modified them before

the current transaction reaches commit point. Without the relaxed semantics such accesses/modifications by other transactions would have caused the current transaction to fail to commit and retry. Fewer retried transactions implies correspondingly reduced stalling in concurrent threads.

6.1.2 Coordinating Execution among Long-Running Concurrent Transactions

Problem Conflicts between long running transactions can be reduced by the previous mechanism. However, in game programming, threads often work collaboratively and can benefit from adjusting their execution based on the execution status of certain other transactions. Traditional STM semantics do not allow any visibility inside a currently executing transaction. This is because an STM transaction has the semantics of executing "all-at-once" at its commit point. In practice, this can cause concurrent threads in games to perform redundant computations if they contain many long running transactions.

Opportunity Any solution to this problem cannot compromise the "all-at-once" execution semantics of transactions, without also compromising the ease-of-programming and verification benefits provided by transactions. However, even a hint saying that another transaction has made *at-least* so much progress can be quite useful for a given transaction to adjust its execution. This adjustment is purely speculative, since there is no guarantee that the other transaction will commit. Subsequently, the thread running the current transaction may have to execute recovery code (such as perform a computation that had been speculatively skipped by the current transaction because the other transaction had already done that computation, but could not commit it).

In domains like gaming, speculative optimizations that are correct with high probability are quite valuable for obtaining high game performance. The communication of such progress hints to other threads can be made best effort, making their communication very low overhead and non-stalling for both the monitored and monitoring

transactions.

Our Solution Using *Progress Indicators*, the programmer can mark lexical program points whose execution progress may be useful to other transactions. Every time control-flow passes a Progress Indicator point, a progress counter associated with that point is incremented. The increments to progress indicators are periodically pushed out globally to make them visible to other transactions that may be monitoring them. However, the RSTM semantics make no guarantees on the timeliness with which each increment will be made visible to monitoring transactions. Each monitoring transaction may have a value for a progress indicator that is significantly smaller (i.e., older) than the most current value of that progress indicator in the thread being monitored. Consequently, the monitoring transactions can only ascertain that *at-least* so much progress (quantified in a program specific manner by the value of the progress indicator) has been made. The monitoring transactions may not be able to ascertain *exactly* how far along in execution the monitored transaction currently is.

6.2 RSTM Language Specification

The RSTM language has two sets of constructs to address the two relaxation mechanisms described in Section 6.1. Use of the `Group Consistency` constructs reduces the commit conflicts between concurrent transactions. The `Progress Indicator` constructs allow for a coordinated execution between concurrent long-running transactions in order to reduce redundant computation across concurrently running transactions. These constructs are described in the following subsections.

6.2.1 Group Consistency

Group consistency semantics can be specified by grouping certain shared program variables accessed inside a given transaction. The programmer can declare each group of variables as having one of four possible relaxed semantics. The group is no longer subject to the default atomicity constraints to which all shared variable and memory

accesses are subjected to within a transaction.

6.2.1.1 *Defining groups*

A group is a declarative construct that a programmer can include at the beginning of the code for an RSTM transaction. A group is a collection of named program variables that could be concurrently accessed from multiple threads. The following C code example illustrates how to define groups.

```
extern int a, b, c, d; /* global variables */

int i = ...;

atomic A(i) {
  group (a, b) : consistency-modifier;
  ...
}
```

In this code example, `A` is the label assigned to the transaction by the programmer. Transaction `A` could be running concurrently in multiple threads. The `A(`i`)` representation allows the programmer to refer to a specific running instance of `A`. The programmer is responsible for using an appropriate expression to compute i in each thread so that a distinction between multiple running instances of `A` can be made. For example, if there are N threads, then i could be given unique values between 0 and $N-1$ in the different threads. `A` would refer to *any one* running instance of transaction `A`, whereas `A(`i`)` would refer to a specific running instance. In all subsequent discussion, the label $\texttt{T}_j$ could refer to either form.

6.2.1.2 *Types of Consistency Modifiers*

For the *consistency-modifier* field in the previous code example, the programmer could use one of the following, exemplified in Figure 30:

1. `none` : Perform no consistency checking on this set of variables. Other transactions could have modified any of these variables after the current transaction accessed them, but the current transaction would still commit (provided no other conflicts unrelated to variables `a` and `b` are detected). The effect of this modifier is distinct from techniques such as *early release*. A shared data item accessed by a transaction can be early-released any time between opening the variable for reading and transaction commit. However once a variable is a part of a group for which the `none` consistency modifier applies no consistency is applied for that variable throughout the lifetime of that transaction. Moreover, unlike early release the `none` modifier is declarative, so the STM system does not keep any bookkeeping information (like version numbers etc), for variables that are in consistency groups with this modifier.

2. `single-source (`$\mathtt{T}_1$`, `$\mathtt{T}_2$`, ...)` : The variables `a` and `b` are allowed to be modified by the concurrent execution of exactly one of the named tansactions without causing a conflict at the commit point of transaction `A`. $\mathtt{T}_1$, $\mathtt{T}_2$, etc are labels identifying the named transactions. If `(`$\star$`)` is given instead of transaction names, then the transaction modifying the variables in the group could be any other single transaction, regardless of its label.

3. `multi-source (`$\mathtt{T}_1$`, `$\mathtt{T}_2$`, ...)`: Similar to `single-source`, except that multiple named transactions are allowed to modify any of the variables in the group without causing a conflict at commit point of A.

```
atomic A(i) {
  group (a, b) : single-source (*)
  group (c, d) : multi-source (B, C)
  ...
}
```

Figure 30: Declaring Group Consistency

6.2.2 Progress Indicators

A programmer can declare progress indicators at points inside the code of a transaction. A counter would get associated with each progress indicator. The counter would get incremented each time control-flow passes that point in the transaction. If the transaction is not currently executing, or has started execution but not passed the point for the progress indicator, then the corresponding counter would have the value -1. Each instance of a running transaction gets its own local copies of progress indicators. Other transactions can monitor whether the current transaction is running and how much progress it has made by reading its progress indicators. As mentioned in Subsection 6.1.2, the progress indicator values are only pushed out from the current transaction on a *best-effort* basis. This is to minimize stalling and communication overheads, while still allowing other transactions to use possibly out-of-date values to determine a lower-bound on the progress made by the current transaction.

The following code sample shows how Progress Indicators are specified in a transaction.

```
atomic A(i) {
  for(j=0; j<N; j++) {
    ...
    progress_indicator x;
    if (...)
      progress_indicators y;
  }
}
```

In the preceding example, the progress indicator `x` is incremented in each iteration of the loop. A special progress indicator called `status` is pre-declared for each transaction. `status` $= -1$ implies that the transaction is not running or it aborted, $= 0$ means that it is currently executing, $= 1$ means that the transaction is currently waiting to commit. Updates to the `status` progress indicator are immediately made available to all monitoring transactions as this is expected to be the most important progress indicator they would like monitor.

Progress indicators can be monitored from transactions running in other threads as shown in Figure 31.

```
atomic B {
  if ( A(2).status == 0 && A(2).x <= 50 ) {
     /* do some extra redundant computation */
  }
  else {
     /* speculatively skip redundant computation */
  }
}

/* Now check global state to determine if A(2)
   actually committed its extra computation, or if B did the extra
   computation.

   If neither, then recover by doing the extra computation now
   (hopefully, this will be relatively rare).
*/
}
```

Figure 31: Monitoring Progress Indicators from other Transactions

6.3 Implementation

We implemented our STM system in C++. In this section, we describe the C++ API we provide to the programmer. We also dwell on low-level considerations that motivated our design.

6.3.1 Overview

The RSTM implementation consists of the following parts:

- *STM_Manager* is a unique object that keeps track of all running and past transactions. It also keeps the master book-keeping for all memory regions touched by a transaction. It acts as the contention manager for the RSTM system. This object is the global synchronizing point for all book-keeping information in the system.

- *STM_Transaction* is the transaction object. It provides functions to open variables for read, write-back values and commit.

- *STM_ReadGroup* groups variables that belong to the same read group. Variables within a group have a notion of consistency as defined in Section 6.2.1. STM_ReadGroups are associated with a transaction. STM_ReadGroups are re-created every-time a transaction starts and are destroyed when the transaction commits.

- *STM_WriteGroup* groups variables that have a particular write consistency model associated with them. They are similar to STM_ReadGroup.

6.3.1.1 Design decisions

Given the constraints we had given ourselves in designing this system, certain design decisions had to be made. We explain these here.

Granularity level Our system has been developed for C/C++ and, as such, the granularity level could not be objects. Indeed, since the programmer can potentially access any object or part thereof through pointer arithmetic, linking book-keeping information to objects is difficult. We therefore keep information at the byte level. However, the overhead associated with byte level book-keeping being considerable, we introduce the notion of zones (see Section 6.3.2) to alleviate the problem.

Hierarchical objects The sole STM_Manager object keeps track of the master copy of all the book-keeping information for the entire machine. However, every other object keeps track of a recent copy of the book-keeping information relevant to the memory zones it is touching. This hierarchy in book-keeping information alleviates the problem that could arise from having a central structure that keeps track of all book-keeping information. Requests to the STM_Manager are not as frequent and synchronization only needs to occur when the information is not present in objects closer to the point of request or during a commit.

Consequences for distributed shared memory systems This hierarchical structure also makes our RSTM implementation portable to DSM systems. Indeed, each shared-memory segment could have a local copy of the book-keeping information, adding a hierarchy layer between the STM_Manager and the STM_Transactions. A central book-keeper is still required to synchronize all the information but intermediate book-keepers are allowed. This is particularly interesting for architectures like IBM's Cell [95] processor.

Transaction roll-back In our implementation, we decided to forgo the use of an undo-log; we used temporary buffer space for write-backs. Although [118] showed that buffering is slower than undo-logs, we believe that buffering has advantages on distributed memory systems. Although we are not demonstrating our work on

these type of architectures, it is likely that STM will have to be developed on these systems. The Cell [95] processor from IBM is an example of such an architecture. In particular, buffering is cheaper to implement on a DSM system because rolling-back a written value means a huge synchronization overhead. Temporary buffers ensure that synchronization only occurs when the value should become visible to other threads, and as such, only occurs once at the commit point of the transaction.

6.3.2 Zone-based management

In our implementation, we introduce the notion of zoned management which help relieve the storage overhead associated with book-keeping at a byte level. We also propose some interesting optimizations to the runtime to allow it to prioritize transactions and intelligently manage transaction commits.

6.3.2.1 Motivation

Our STM system was written with games in mind. Games are usually written in C/C++ and make heavy use of direct memory access. As such, object-based management was not an option as data stored in an object can be accessed directy through pointers, thus bypassing any book-keeping information stored in the object. We thus decided to manage our book-keeping at the byte level (which is the smallest addressable entity in C). It became quickly apparent that maintaining book-keeping information at the byte level would use up too much memory and storage space. For each byte we would need to keep track of the version number (4 bytes) and an identifier for the last transaction that wrote to that byte (another 4 bytes). In total, for every byte of memory accessed via a transaction, we would need to keep track of 8 bytes of information. We also quickly realized that most of the information would be redundant. For example, modifying an int results in the modification of 4 consecutives bytes of data but all 4 bytes have the same metadata (version number and last transaction information). We thus decided to store information at a zone level. Each

byte can be individually queries for its metadata but it is not stored for each byte.

6.3.2.2 Definition of a zone

A zone is defined as a contiguous section of memory with the same metadata. Metadata, in our case, is the version number and the information regarding the last transaction that wrote to the memory region. Zones dynamically merge and split to maintain the following two invariants:

- All bytes within a zone have the same metadata.
- Two zones that are contiguous but separate differ in metadata.

The first invariant guarantees correctness because the properties of an individual byte are well-defined and easily retrievable. The second invariant guarantees that the book-keeping information will be as small as possible. Section 6.3.4 explains how commits are implemented to try to generate as few zones as possible.

Note that our notion of zones is different from that of orecs [88]. Zones are an implementation mechanism destined to compress the total information stored for book-keeping. They have no implication on the functionality of the STM. To the user, the use of zones or the use of a byte-level book-keeping is equivalent. The same information can be obtained in both cases. On the other hand, orecs are used by the STM logic and need to be obtained by transactions before they may read/write to a memory word. They control which transactions can read/write or otherwise modify a particular memory word. In our case, zones have no such logic and are merely a book-keeping artifact.

6.3.3 API overview

In this section, we will describe the API for the main classes of our system.

6.3.3.1 *STM_MemoryManager*

The API provided by the STM_MemoryManager allows zone management of the memory. The API provides the following access points:

- *Retrieve properties for a zone.* The programmer can request the version and last writer of any arbitrary zone of memory. The zone can be one byte or it can be a larger piece of contiguous memory. It does not have to match zones used internally to represent the memory.
- *Set properties for a zone.* Similarly, properties such as version number and last writer can be set for any arbitrary zone of memory.
- *Zones query.* Allows the programmer to determine whether a zone is being tracked or not.

Thus, the API allows for a view of memory at a byte level while maintaining information at a zone level. The exact way in which information is stored is abstracted away from the programmer.

6.3.3.2 *STM_Manager*

The STM_Manager object provides three main functions to the user as shown in Figure 32. The 'getTransaction' function will return a transaction corresponding

```
1 STM_Transaction *getTransaction(uint id);
  list<uint> getVersionAndLock(void *location, uint size, uint id);
3 void unlock(void *location, uint size, uint id);
```

Figure 32: API for the STM_Manager

to a given ID. The STM_Manager needs to know about transactions as it needs to know about which transactions may potentially commit in order to perform certain optimizations. This is the reason why transaction objects are obtained from the

STM_Manager directly. The other two functions are used when committing transactions. When a transaction commits, it has to atomically check if anyone has written to where it wants to write and lock the location. When a transaction has obtained a lock on a memory location, any other transaction trying to write back its value to that zone will fail and have to either wait or retry. This thus guarantees that all the writes from a given transaction occur atomically with respect to writes from other transactions.

6.3.3.3 STM_Transaction

The STM_Transaction object implements the main functionalities common in all STM systems. It further adds support for relaxed semantics. The main API is described in Figure 33. The 'openForRead' function opens a variable for reading and puts it in

```
1 void commit();
  void openForRead(void *loc, uint size, list<STM_ReadGroup*> groups);
3 void writeBack(void *loc, uint size, void *data, STM_WriteGroup*
  group);
```

Figure 33: API for STM_Transactions

the specified STM_ReadGroups. The groups are then responsible for enforcing their particular flavor of consistency. The 'writeBack' function opens a variable for write and buffers the write-back. 'commit' will try to commit the transaction by checking if all the read groups can commit and if the variables can be written back correctly.

6.3.3.4 STM_ReadGroup

The STM_ReadGroup allows specification of the majority of the relaxed semantics. The programmer can specify the type of consistency a read group will enforce.

6.3.4 Operational aspect of commits

The commit of a relaxed transaction is very similar to that of a regular transaction. However, certain consistency checks are skipped due to relaxation in the model. The

following steps are performed when commiting a transaction. In the following the term "modified" refers to the write to a variable when some transaction commits.

- Check to make sure if the default read group can commit. This group enforces traditional consistency for all variables that are not part of any other group. Therefore, all variables in the default group must not have been modified between the time they are read and the time the transaction commits.
- Check to make sure if read groups can commit. This will implement the relaxed consistency model previously discussed. Read groups can commit under certain conditions even if the variables they contain have been modified.

Committing a read group Committing a read group is simply a matter of enforcing the consistency model of the group on the variables present in the group. Checks are made on each zone that is present in the read group to see if they have been modified, and, if they have, if it is still correct to commit given the relaxed consistency model.

Committing a write group Committing a write group consists of:

- acquiring a lock from the STM_Manager on all locations the group wants to update;
- checking to make sure that there were no intermediate writes;
- writing back the buffered data to the actual location;
- updating the version and owner information for the locations updated;
- unlocking the locations and releasing the space aquired by the buffers (now useless).

Write groups can also still presume that they have successfully committed even if there was a version inconsistency provided that it was within the bounds indicated by the relax consistency model. Note that in the case of a version mismatch that is acceptable, the buffered value is not written back.

Zones and committing Since we have a zone-based book-keeping scheme, we want to minimize the number of zones. Therefore, when a write group commits, it will set the version of all the zones it is committing to the same number. This new version number will be greater than all the old version number for all the zones being updates. This ensures correctness also allows for the minimization of the number of zones that will be used for the write group. Since the properties for the zones are all the same (same last writer and same version), all contiguous zones will be merged. While this may not be the optimal solution to globally obtain the minimum number of zones, it does try to keep the number of zones low.

6.3.5 Runtime Optimizations

We implement some prioritization based optimization in the runtime. The basic idea is that transactions will higher priority and a near completion time should be allowed to commit before transactions with a lower priority that may already be trying to commit. The STM_Manager will try to factor this into account. It does this by stalling the call to 'getVersionAndLock' of a lower priority thread A if the following two conditions are met:

- A higher priority thread (B) has segments intersecting with those of A
- B is close to committing.

It will thus let the other transaction (B) commit and then will allow A to proceed. A timeout mechanism is also present to prevent complete lack of forward progress.

Table 6: Table showing the number of Aborts, Commits, Transaction Throughput in Transactions per second and the ratio of the Transaction Throughput and the Theoretical Peak Transaction Throughput. P is the number of particles in the system and N is the number of threads.

Metrics	**Baseline** (P/N)				**RSTM** (P/N)			
	1024/16	2048/32	4096/32	4096/64	1024/16	2048/32	4096/32	4096/64
# of Aborts	128	250	231	454	124	235	434	369
# of Commits	160	320	320	640	160	320	320	640
Throughput(TT)	7.266	12.79	6.98	25.16	16.30	27.70	12.02	33.57
% of Peak	0.428	0.445	0.533	0.666	0.961	0.964	0.918	0.888

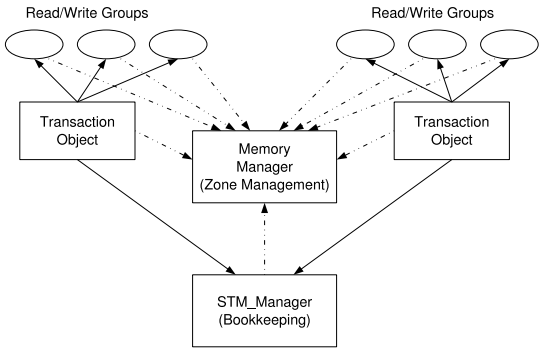


Figure 34: Incremental Communication

6.4 Results

In this section we evaluate the relaxed consistency model, and show that for applications that need only a minimum acceptable consistency, applying the relaxed consistency model results in significantly less number of aborted transactions and hence increased transaction throughput.

6.4.1 A dynamic particle system

We evaluate our model using an application that simulates particle systems and we give some details about the nature of these systems. Particle systems provide for the creation and evolution of complex structure and motion of particles, from a relatively small set of rules [123]. Such systems have been used in diverse scenarios ranging from stochastic modeling, molecular physics to real-time simulation and computer gaming [110, 111]. Particle systems have been widely studied in the context of parallelization. The specific particle system implemented in our RSTM consists of a number of particles distributed among a number of threads (one thread processes one block of particles). Each particle has a position vector, a velocity vector and a mass associated with it. Each of the particles experiences two forces - a constant force (such as gravity) and also the gravitational force between pairs of particles. The

system evolves in timesteps and, at each timestep, the movement of the particles due to these forces is computed using numerical integration methods. Specifically, Euler integration is used to calculate the values of the position and velocity attributes of a particle p using the following equation:

$F_p(t + dt) = F_p(t) + dt * F_p'(t)$

where F_p represents either the velocity or position of the particle p. While most particle systems are parallelizable, they are not embarrassingly so because of interactions between particles that are being processed separately. As a simple example, the $\overrightarrow{Velocity_p}$ calculated for particle p in timestep $t + dt$, depends on the $\overrightarrow{Force_p}$ acting on the particle in timestep $t + dt$, which in turn depends on the distance vectors from particle p to all other particles in timestep t (Laws of Gravitation). This would normally incur serialization.

6.4.2 Relaxation

We briefly describe the algorithm for the particle system simulation benchmark in the following. Each of the timesteps should result in exactly one set of updates to the particles' attributes. This is placed in the body of an atomic block, and the current timestep or iteration count is exported as a Transaction State. The transaction T_i declares the particle attributes of its neighboring transactions T_{i-1} and T_{i+1} to be in its read-group. It then uses these values to compute the new attributes of its own particles. Finally, it tries to commit these values and if a consistency violation is detected, it aborts and retries. The intuition to the relaxation of consistency here is that particles that are far away from a particle p, do not exert much force on it whereas particles in the blocks neighboring that of p, do exert a significant force on p. Thus, in the calculation of the force vector for each p in block i, read consistency is followed only when reading positions of particles in neighboring blocks $i - 1$ and $i + 1$. Even though the positions of particles in other blocks are also read, they are not added to

a ReadGroup and hence are not check for consistency violation at commit time, since reading somewhat stale positions of such distant particles will not affect the accuracy of $\overrightarrow{Force_p}$ by much. Also, even for nearby particles, the relaxation model accepts a certain staleness (one timestep ahead or behind). This relaxation is achieved by using the progress indicators and group consistency modifiers. Each transaction updates its progress indicator at the boundary of each time step. A transaction wishing to read the particle positions owned by another transaction will add the latter to its group consistency transaction list. If the the producer transaction is the owner of a cell close to the one owned by the consumer transaction, the producer is added to the group consistency list with the `single-source` or `multi-source` modifiers.

6.4.3 Experimental Evaluation

We implemented the particle system described above using the RSTM constructs and APIs described in Section 6.2 on a machine with an Intel Pentium Core2Duo processor with 4 cores, 1GB of main memory running RedHat Enterprise Linux 3. In order to compare RSTM operation semantics with those of a conventional STM, we operate the RSTM in strict consistency mode, where atomicity is preserved and read/write group violations are guaranteed to result in an abort. This is our baseline case. In the relaxed STM mode, as described above, the group consistency models are used and strict checking read/write violations is not enforced. Using both modes of operation, we measured several metrics like the total number of aborts and the total number of commits. In addition, we also measured the transaction throughput (TT) which is the number of transactions committed per second. Moreover, we also measured the theoretical peak transaction Throughput which is the number of transactions that can commit per second if the STM model did not enforce any atomicity constraints at all. The program would obviously produce incorrect results, but this number would represent the upper bound on the throughput that can be achieved by any STM for

this application. The results are summarized in Table 6. From the table, we notice that the number of aborts are lower in the RSTM in two cases, but are twice that of the baseline STM in the third case. However, the transaction throughput in the third case is much higher than in the first two cases.

6.5 Related work

STMs have been studied for over ten years now. First introduced by Shavit and Touitou in 1995 [122], language extensions have been proposed to implement STM in Haskell [89], Caml [115], and Java [73, 94] among others. Many optimizations to STM and its implementations have been proposed, the most recent being [90, 66]. However, our work has unique contributions not brought by any of the previous papers.

6.5.1 Relaxed consistency

Relaxed consistency has already been studied in other contexts. For example, in parallel computing, Zucker studied relaxed consistency and synchronization [132]. Adve also looked at the ordering of read and write events in a system to provide a weaker ordering [67]. In [77], a primitive called the early release construct is proposed, in order to allow uncommitted transactions to share their results with other transactions. Several other works [106, 108, 85] have noted that in many domains, it may be acceptable to compute an approximate solution, that is, allow an amount of imprecision if it helps to reduce runtime or leads to better task schedules at runtime [81]. Although flexible and loose consistency STMs have been proposed before, to our knowledge none have offered to the programmer the flexibility to statically or dynamically control the amount of consistency - the set of variables for which consistency matters and the set for which it doesn't. Our notion of read-groups where consistency is defined on a group by group basis. These semantics allow the programmer to fully express thread interactions. We showed that such a feature is a very powerful programming tool especially in the multimedia and gaming domains.

6.5.2 C/C++ language extension

Although [73] proposes a language extension to support transactional memories, it extends the Java language. Most language extensions implementing STM are not for the C/C++ language. The C++ language, while offering all the object abstractions that Java provides, also allows direct access to the memory. This added possibility makes defining the granularity at which STM operates a problem. If one uses an object-level granularity, one cannot be certain that another transaction will not access the object's memory location through a pointer arithmetic operation. The obvious granularity for C/C++ languages is thus the byte although this causes more overhead. Our approach of building zones in memory allows for a byte-level granularity while keeping the book-keeping overhead acceptable. The zone-level granularity could be extended to take into account more complex layouts of memory and thus deal with non-contiguous zones. Although zone-level memory management is not novel in itself, we have applied it to STM and our commit stage tries to actively minimize the number of zones required. Other optimization algorithms could be implemented to try to compact the zone representation due to the fact that the exact value of a version number is not important, just its relative magnitude (version numbers should never decrease).

6.6 Conclusion

In this work we propose an extension to the Software Transactional Memory model that relaxes the traditional consistency semantics between transactions. The relaxed semantics more naturally capture the interaction constraints between threads in application domains like gaming and multimedia. The relaxed STM (RSTM) model allows for better scaling of application performance to a large number of cores because the relaxed semantics causes significantly lower serialization between transactions.

We adapted a parallel particle simulation application to use RSTM transactions for

inter-thread communication of particle attributes (positions and velocities). Adapting the application for RSTM was simply a matter of wrapping each of the existing critical sections in our RSTM atomic regions, and specifying a relaxed consistency criterion. The relaxed consistency criterion allowed the simulations of individual particles to use slightly older attributes for some of the other particles. Our results demonstrate that the use of RSTM provides a tremendous increase in application performance over a traditional STM model. This was due to a significantly reduced serialization of transactions using the RSTM model and a lower number of transaction aborts.

CHAPTER VII

CONCLUSION

While transactional systems are receiving significant attention as a programmer-friendly solution to problems in parallel programming, their performance has been shown in studies to be lagging behind that of fine-grained locking. This performance disparity significantly offsets the programmability advantages of the TM model. This thesis described techniques and methods for reducing the cost of using transactions for synchronizing shared data between critical sections. These range from checkpointing and conflict recovery mechanisms to a hybrid optimistic-pessimistic irrevocable transaction model to a relaxed-consistency programming and execution models that allow significantly higher transaction throughput.

In addition to a disparity in performance, transactional memory systems also carry from a programming model that is too constrained to be able to express many commonly occuring programming idioms. This thesis asks whether the database-style TM semantics and programming models are appropriate for modern, emerging parallel applications. Specifically it discusses three phenomena "Approximate Sharing", "Fine-Grained Consistency" and "User defined recovery" that are important to such programs and it discusses the difficulty of using traditional transactional memory programming models to express these idioms. Programmability, while desirable, should not necessarily result in restricting the kinds of semantics that can be expressed in parallel programs.

Finally, despite this recent flurry of interest in transactional memory systems and the significant amount of literature most of the studies investigating the use and optimization of these systems have been limited to smaller benchmarks and suites

containing small to moderate sized programs which makes translating these insights to large real-world parallel programs difficult. In this thesis we described the parallelization of a large, real-time, interactive rigid body phyics engine that is used in hundreds of commercial games. The lessons learnt in this investigation suggest some areas of improvement to the TM programming model such as providing safe methods for conditional waiting and conditional synchronization and the importance of compiler support in guaranteeing safety especially in the presence of third-party libraries. In addition this investigation also highlighted a few fundamental obstacles in parallelizing large sequential legacy code bases such as the need to re-design important data structures to be better suited for disjoint sharing between threads and the big role that domain knowledge plays in extracting parallel performance. Finally, this investigation also highlighted the notion of *algorithmic speculation* and a need for language constructs and a runtime for explicitly expressing such speculation.

7.1 Future Research

Many of the topics investigated in this thesis have extensions that are good starting points for further exploration. Some of these are outlined below.

- Transaction Slicing: The automatically synthesized corrective handlers described in Chapter II are designed to restore the state of a transaction to some prior program point and enable it to make forward progress. If it is possible to statically compute a precise "Transaction Slice", then when a transaction experiences a conflict it restores its state to a previous point in its execution and then instead of executing the entire transaction from that point, only executes the statically computed *slice* of the transaction starting at that point. This enables transactions recover from conflicts by making *localized, surgical* repairs to their state. Apart from the challenge of building an efficient inter-procedural

slicing transformation, this scheme significantly impacts the core corrective recovery mechanism but may produce significant performance improvements in many programs.

- Algorithmic Speculation: Approaches to using data speculation for parallelism have so far been low-level and completely hidden from the programmer. For example the *Thread Level Speculation* (TLS) systems automatically find speculation points in the program (such as branch instructions for example) and being to speculatively execute instructions starting at that point. Transactional Memory systems also leverage speculation but hide it from the application programmer. Is is well known that there is a large class of algorithms that are inherently sequential - there are either no good parallel versions of these algorithms that show good scalability or if they exist, they are very cumbersome to develop. Examples of algorithms in this class are many algorithms from the P-Complete class of problems. We have observed that speculation may offer a novel way towards parallelization of these problems. Specifically our approach consists of a SpeculativeTask{...} construct that is exposed to the programmer to specify work that can be done speculatively in a separate task that runs concurrently with the main non-speculative task. The runtime ensures that all the side-effects from executing this speculative task are contained and multiple speculative tasks do not affect each other - this is similar to the *isolation* guarantee for transactions. Additionally, we also provide constructs to specify when particular speculative tasks can be harvested and when each of them is finished. Briefly, a speculative task uses the following operations:

  - Spec-Start(*state*): This operation starts a new speculative task with the an initial state *state*. Concretely, *state* can be implemented as a series of speculative writes of specific values that set up the speculative tasks initial

conditions.

- Spec-Fold(): This operation *folds* a currently active speculative task into the main non-speculative task and merges the state computed by the speculative task into the state of the main non-speculative task.
- Spec-Stop(): This operation aborts the execution of a currently active speculative task and discards all the state it has computed. This is used for example when the initial *state* used for this speculative start has been invalidated due to writes by some other speculative task that finished or due to concurrent writes in the non-speculative task.

We have used this approach to implement a version of the speculative island discovery algorithm in Chapter V, Section 5.4. As seen in the plot in Figure 29, this form of speculative parallelization for this algorithm yields a speedup of upto 4.2X and moreover scales well. There are instances of other well-known P-Complete problems from [87] that appear suitable for parallelization using this approach. Additionally, it appears that this approach can also be used for constructing a programming model for adaptive, speculative domain decomposition for parallel programs.

REFERENCES

[1] Dice D., Shalev O., Shavit N., "Transactional Locking II". In *proceedings of the 20th International Symposium on Distributed Computing* (DISC), Stockholm, Sweeden, Sept. 2006.

[2] Acar U.A., Hammer M.A., Chen. Y, "CEAL: A C-Based Language for Self-Adjusting Computation". In *Proceedings of the International Conference on Programming language design and implementation* (PLDI) 2009.

[3] Minh C.C., Chung J., Kozyrakis K., Olukotun K., "STAMP: Stanford Transactional Applications for Multi-Processing". In *IISWC 2008* 35-46

[4] Shavit, N., Touitou, D. Software transactional memory. PODC 95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing (New York, NY, USA, 1995), ACM Press, pp. 204213.

[5] Guerraoui R., Herlihy M., Pochon B., "Toward a Theory of Transactional Contention Managers". In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing* Las Vegas, NV, Aug. 2005.

[6] Kulkarni M., Pingali K., Walter B., Ramanarayanan G., Bala K., Chew L.P., "Optimistic parallelism requires abstractions". In the *International Conference on Programming Language and Design* 2007

[7] Felber P., Fetzer C., and Riegel T., "Dynamic Performance Tuning of Word-Based Software Transactional Memory". In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* 2008

[8] Jaswanth S., Pande S., "Exploiting Approximate Value Locality for Data Synchronization on Multicore Processors". In the Proceedings of the *IEEE International Symposium on Workload Characterization (IISWC)* 2010.

[9] Herlihy M., Luchangco V., Moir M., and Scherer W.N., "Software transactional memory for dynamic-sized data structures". In *Proceedings of the twenty-second annual symposium on Principles of distributed computing* (PODC '03)

[10] Phatak, S.H. and Badrinath, B.R. "Multiversion Reconciliation for Mobile Databases" in Proceedings of the 15th international Conference on Data Engineering 1999.

[11] Lipasti, M. H., Wilkerson, C. B., Shen, J. P. Value locality and load value prediction. SIGOPS Oper. Syst. Rev. 30, 5 (Dec. 1996), 138-147

[12] Minh C.C., Chung J., Kozyrakis K., Olukotun K., "STAMP: Stanford Transactional Applications for Multi-Processing". In *IISWC 2008* 35-46

[13] Gray, J. N., Lorie, R. A., Putzolu, G. R., and Traiger, I. L. 1988. Granularity of locks and degrees of consistency in a shared data base. In Readings in Database Systems Morgan Kaufmann Publishers, San Francisco, CA, 94-121.

[14] W. N. Scherer III and M. L. Scott, "Advanced Contention Management for Dynamic Software Transactional Memory" in Proc. of the 24th ACM Symp. on Principles of Distributed Computing, Las Vegas, NV, July 2005

[15] The LLVM Compiler Infrastructure, www.llvm.org

[16] Richardson S.E., Exploiting Trivial and Redundant Computation, Sun Microsystems Technical Report 1993.

[17] Ni, Y., Menon, V. S., Adl-Tabatabai, A., Hosking, A. L., Hudson, R. L., Moss, J. B., Saha, B., and Shpeisman, T. 2007. Open nesting in software transactional

memory. In Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (San Jose, California, USA, March 14 - 17, 2007)

[18] Olszewski, M., Cutler, J., and Steffan, J. G. 2007. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In Proceedings of the 16th international Conference on Parallel Architecture and Compilation Techniques (September 15 - 19, 2007)

[19] Richardson S. E., Caching Function Results: Faster Arithmetic by Avoiding Unnecessary Computation Sun Microsystems Technical Report TR-92-1 1992.

[20] T. Harris and S. Stipic "Abstract Nested Transactions", in 2nd Workshop on Transactional Computing (TRANSACT 07).

[21] Yang, J. and Gupta, R. 2002. Frequent value locality and its applications. Trans. on Embedded Computing Sys. 1, 1 (Nov. 2002)

[22] Lepak, K. M., Exploring, Defining, and Exploiting Recent Store Value Locality. Ph.D. thesis. The University of Wisconsin-Madison, Department of Electrical and Computer Engineering. Dec. 2003.

[23] Ramadan, H. E., Roy, I., Herlihy, M., Witchel, E, "Committing Conflicting Transactions in an STM", in Proc. of the International Symposium on the Principles and Practice of Parallel Programming (PPOPP) 2009.

[24] Acar U. A., Blelloch G. E., and Harper R. 2003. Selective memoization. SIGPLAN Not. 38, 1 (Jan. 2003)

[25] Ni Y., Menon V., Adl-Tabatabai A. R, Hosking A.L, Hudson R.L., Moss J.E.B., Saha B., Shpeisman T., Open nesting in software transactional memory. Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP):68-78 March 2007.

[26] Eliot, J., Moss, B., Open nested transactions: Semantics and support. In Workshop on Memory Performance Issues 2005.

[27] M. Herlihy and E. Koskinen "Checkpoints and continuations instead of nested transactions", in the 3rd Workshop on Transactional Computing

[28] Adya, A. 1999 Weak Consistency: a Generalized Theory and Optimistic Implementations for Distributed Transactions. Technical Report. UMI Order Number: TR-786., Massachusetts Institute of Technology.

[29] Dmitri Perelman and Idit Keidar, "On Avoiding Spare Aborts in Transactional Memory", in Proc. 21st Symposium on Parallelism in Algorithms and Architectures (SPAA) 2009.

[30] Binkley D., "Precise executable interprocedural slices". In *ACM Letters on Programming Languages and Systems* 2, 1-4 (March 1993)

[31] Guerraoui R., and Kapalka M., On the correctness of transactional memory. In *Principles and Practice of Parallel Programming* (PoPP) 2008.

[32] Moss, J. E. and Hosking, A. L. 2006. "Nested transactional memory: model and architecture sketches". In *Science of Computer Programming* 63, 2 (Dec. 2006), 186-201

[33] Blundell C., Raghavan A., Martin M.K., "RETCON: transactional repair without replay". In *Proceedings of the 37th annual international symposium on Computer architecture* (ISCA '10)

[34] Herlihy, M. and Koskinen, E., "Transactional boosting: a methodology for highly-concurrent transactional objects". In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* 2008.

[35] Ramalingam G. and Reps T., A Categorized Bibliography on Incremental Computation. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages* pages 502510, 1993.

[36] Bieniusa A., Middelkoop A., Thiemann P., "Actions in the Twilight: Concurrent irrevocable transactions and Inconsistency repair". Technical Report 257, Insitut fr Informatik, Universitt Freiburg, 2010

[37] Pugh, W. and Teitelbaum, T., "Incremental computation via function caching". In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* 1989.

[38] Lepak, M.L., "Exploring, Defining, and Exploiting Recent Store Value Locality" *Ph.D. thesis*, University of Wisconsin-Madison, Department of Electrical and Computer Engineering. Dec. 2003

[39] Yellin, D. M., Strom, R. E., "INC: a language for incremental computations" *ACM Transactions on Programming Languages and Systems* 13, 2 (Apr. 1991), 211-236.

[40] Sundaresh, R. S. and Hudak, P., "A theory of incremental computation and its application". In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* 1991.

[41] Citron D. and Feitelson D. G.,"Look It Up" or "Do the Math": An Energy, Area, and Timing Analysis of Instruction Reuse and Memoization, in Workshop on Power Aware Computing Systems 2004.

[42] Zadeh L.A., Fuzzy logic, neural networks, and soft computing. Communications of the ACM, 37(3):7784, 1994.

[43] De Gelas, J. The quest for more processing power: Multi-core and multi-threaded gaming. http://www.anandtech.com/cpuchipsets/showdoc.aspx, March 2005.

[44] Baek W., Chung J., Minh C. C., Kozyrakis C., and Olukotun K., Towards soft optimization techniques for parallel cognitive applications. In Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (San Diego, California, USA, June 09 - 11, 2007). SPAA '07. ACM, New York, NY, 59-60.

[45] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. 2006. Autolocker: synchronization inference for atomic sections. In Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '06).

[46] Yeh T. Y., Reinman G., Patel S. J., and Faloutsos P. 2009. Fool me twice: Exploring and exploiting error tolerance in physics-based animation. ACM Trans. Graph. 29, Dec. 2009

[47] Yeh T., Faloutsos P., Ercegovac M., Patel S. and Reinman G. 2007. The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration. In Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture (December 01 - 05, 2007).

[48] Open Dynamics Engine, http://ode.org

[49] R. Guerraoui, M. Kapalka, J. Vitek, STMBench7: A benchmark for software transactional memory, in *Proceedings of the 2nd European systems conference, Mar. 2007*

[50] M. J. Carey, D. J. DeWitt, C. Kant, J. F. Naughton, A status report on the OO7 OODBMS benchmarking effort In *OOPSLA 94: Proc. 9th annual conference*

on object-oriented programming systems, language, and applications, pages 414426, Oct. 1994.

[51] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, Anoop Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*

[52] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Lujan, Chris Kirkham, Ian Watson, Lee-TM: A Non-trivial Benchmark for Transactional Memory in *Proc. 7th International Conference on Algorithms and Architectures for Parallel Processing 2008*

[53] Gokcen Kestor, Srdjan Stipic, Osman S. Unsal, Adrian Cristal, Mateo Valero, RMS-TM: A Transactional Memory Benchmark for Recognition, Mining and Synthesis Applications In *4th Workshop on Transactional Computing (TRANSACT) 2009*

[54] D. Harmanci, P. Felber, M. Sukraut, and C. Fetzer, TMunit: A transactional memory unit testing and workload generation tool *Technical Report RR-I-08-08.1, Universite de Neuchatel, Institute dInformatique, Aug. 2008*

[55] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman, "Compiler and runtime support for efficient software transactional memory" In *Proc. 2006 ACM SIGPLAN conference on programming language design and implementation, pages 2637, June 2006*

[56] James Reinders, Intel Threading Building Blocks, O'Reilly Media 2007.

[57] Tim Sweeney, The Next Mainstream Programming Language: A Game Developers Perspective, Invited Talk at the *International Symposium on Principles of Programming Languages 2006*

[58] S. Brown, S. Attaway, S. Plimpton, and B. Hendrickson, Parallel strategies for crash and impact simulations, in *Computer Methods in Applied Mechanics and Engineering* 184:375390, 2000.

[59] Grinberg, I. and Wiseman, Y., Scalable parallel collision detection simulation in *Proceedings of the Ninth IASTED international Conference on Signal and Image Processing* 2007

[60] Lawlor, O. S., Chakravorty, S., Wilmarth, T. L., Choudhury, N., Dooley, I., Zheng, G., and Kal, L. V. 2006, ParFUM: a parallel framework for unstructured meshes for scalable dynamic physics applications, in *Engineering with Computers* Dec. 2006

[61] M. Figueiredo, T. Fernando, An Efficient Parallel Collision Detection Algorithm for Virtual Prototype Environments in the *10th International Conference on Parallel and Distributed Systems* 2004.

[62] Tang, M., Manocha, D., and Tong, R. Multi-core collision detection between deformable models. In *SIAM/ACM Joint Conference on Geometric and Physical Modeling* 2009

[63] Ferad Zyulkyarov, Vladimir Gajinov, Osman Unsal, Adrin Cristal, Eduard Ayguad, Tim Harris, Mateo Valero, Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server in *14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)* Feb 2009

[64] O.S Lawlor, L.V. Kale, A voxel-based parallel collision detection algorithm in *Proceedings of the 16th international conference on Supercomputing 2002*

[65] C. Addison, Y. Ren, and M. van Waveren. Openmp issues arising in the development of parallel blas and lapack libraries. *Scientific Programming*, 11(2):95 – 104, 2003.

[66] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1133985.

[67] S. V. Adve and M. D. Hill. Weak ordering—A new definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 2–14, 1990. URL `citeseer.ist.psu.edu/adve90weak.html`.

[68] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus ipc: the end of the road for conventional microarchitectures. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 248–259, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-232-8. doi: http://doi.acm.org/10.1145/339647.339691.

[69] Kunal Agrawal, Yuxiong He, Wen Jing Hsu, and Charles E. Leiserson. Adaptive scheduling with parallelism feedback. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 100–109, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-189-9. doi: http://doi.acm.org/10.1145/1122971.1122988.

[70] Guillem Bernat, Antoine Colin, and Stefan M. Petters. Wcet analysis of probabilistic hard real-time systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 279, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1851-6.

[71] L. Bishop, D. Eberly, T. Whitted, M. Finch, and M. Shantz. Designing a pc game engine. *IEEE Computer Graphics and Applications*, 18(1):46–53, January

1998. ISSN 0272-1716.

[72] Hans-J. Boehm. Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 261–268, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-056-6. doi: http://doi.acm.org/10.1145/1065010.1065042.

[73] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The atomos transactional programming language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1133983.

[74] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous subordinate microthreading (ssmt). In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pages 186–195, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0170-2. doi: http://doi.acm.org/10.1145/300979.300995.

[75] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05*, pages 519–538, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-031-0. doi: http://doi.acm.org/10.1145/1094811.1094852.

[76] S. Costa. Game engineering for a multiprocessor architecture. Master's thesis, John Moores University, Liverpool, 2004.

[77] Cray. Chapel specification, February 2005.

[78] David E. Culler, Andrea C. Arpaci-Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine A. Yelick. Parallel programming in split-c. In *Supercomputing*, pages 262–273, 1993. URL citeseer.ist.psu.edu/article/culler93parallel.html.

[79] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science and Engineering, IEEE*, 5(1):46 – 55, 1998. ISSN 1070-9924.

[80] Johan de Gelas. The quest for more processing power: Multi-core and multi-threaded gaming. http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2377\&p=3, March 2005.

[81] Romulo Silva de Oliveira, Joni da Silva Fraga, and Jean-Marie Farines. Scheduling imprecise tasks in real-time distributed systems. In *ISORC '01*, pages 319–326. IEEE Computer Society, 2001. doi: http://doi.ieeecomputersociety.org/10.1109/ISORC.2001.922855.

[82] Norman Draper and Harry Smith. *Applied Regression Analysis*. Wiley Series in Probability and Statistics, 2000. ISBN 0471170828.

[83] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd ed.)*. Wiley Interscience, 2000. ISBN 0-471-05669-3.

[84] Alexandre E. Eichenberger, Kevin O'Brien, Kathryn M. O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, Michael Gschwind, Roch Archambault, Yaoqing Gao, and Roland Koo. Using advanced compiler technology to exploit the performance of the cell broadband engine$^{\text{tm}}$ architecture. *IBM Systems Journal*, 45(1):59–84, 2006.

[85] Wu-Chun Feng. Applications and extensions of the imprecise-computation model. Technical Report UIUCDCS-R-96-1951, University of Illinois at Urbana Champaign, 1996. URL `citeseer.ist.psu.edu/81344.html`.

[86] Didier Le Gall. Mpeg: a video compression standard for multimedia applications. *Commun. ACM*, 34(4):46–58, 1991. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/103085.103090.

[87] Raymond Greenlaw , H. James Hoover , Walter L. Ruzzo, "A Compendium of Problems Complete for P", University of Alberta Tech. Report TR91-11 1991.

[88] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-712-5. doi: http://doi.acm.org/10.1145/949305.949340.

[89] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-080-9. doi: http://doi.acm.org/10.1145/1065944.1065952.

[90] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 14–25, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-320-4. doi: http://doi.acm.org/10.1145/1133981.1133984.

[91] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Bull.*, (37): 28–29, 1972. ISSN 0163-5719. doi: http://doi.acm.org/10.1145/1056777.1056779.

[92] Rolf Hempel. The mpi standard for message passing. In *HPCN Europe 1994: Proceedings of the nternational Conference and Exhibition on High-Performance Computing and Networking Volume II*, pages 247–252, London, UK, 1994. Springer-Verlag. ISBN 3-540-57981-8.

[93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM Press. ISBN 0-8186-3810-9. doi: http://doi.acm.org/10.1145/165123.165164.

[94] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-708-7. doi: http://doi.acm.org/10.1145/872035.872048.

[95] H. Peter Hofstee. Power efficient processor architecture and the cell processor. In *HPCA '05*, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2275-0. doi: http://dx.doi.org/10.1109/HPCA.2005.26.

[96] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. 2010. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'10)

[97] IdSoftware. `ftp://ftp.idsoftware.com`. idSoftware download site.

[98] Intel Road Map. http://www.intel.com/cd/ids/developer/asmo-na/eng/201969.htm?page=1, 2006.

[99] Kaneva. http://www.kaneva.com. Kaneva website.

[100] Dongkeun Kim and Donald Yeung. Design and evaluation of compiler algorithms for pre-execution. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 159–170, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-574-2. doi: http://doi.acm.org/10.1145/605397.605415.

[101] Dongkeun Kim and Donald Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Trans. Comput. Syst.*, 22(3):326–379, 2004. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/1012268.1012270.

[102] Hyesoon Kim, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. 2d-profiling: Detecting input-dependent branches with a single input data set. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 159–172, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2499-0. doi: http://dx.doi.org/10.1109/CGO.2006.1.

[103] Bil Lewis and Daniel J. Berg. *Multithreaded programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998. ISBN 0-13-680729-1.

[104] Michael Lewis. The new cards. *Commun. ACM*, 45(1):30–31, 2002. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/502269.502289.

[105] Michael Lewis and Jeffrey Jacobson. Introduction. *Commun. ACM*, 45(1): 27–31, 2002. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/502269.502288.

[106] J. Liu, K. Lin, R. Bettati, D. Hull, and A. Yu. Use of imprecise computation to enhance dependability of real-time systems, 1994. URL citeseer.ist.psu.edu/liu94use.html.

[107] Alexander Maksyagin. *Modeling Multimedia Workload for Embedded System Design*. PhD thesis, Swiss Federal Institue of Technology, Zurich, 2005. ETH No. 16285.

[108] N. Malcolm and W. Zhao. Version selection schemes for hard real-time communications. In *Proceedings of Real-Time Systems Symposium*, pages 12–21, December 1991.

[109] Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 27, pages 127–144, New York, NY, 1992. ACM Press. URL citeseer.ist.psu.edu/article/masuhara94objectoriented.html.

[110] D. McAllister. The design of an api for particle systems, 2000. URL citeseer.ist.psu.edu/mcallister00design.html.

[111] Serge Miguet and Jean-Marc Pierson. Dynamic load balancing in a parallel particle simulation. In *High Performance Computing Symposium*, pages 420–431, 1995. URL citeseer.ist.psu.edu/miguet95dynamic.html.

[112] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Matthew Iyer, Dan Fay, David Hodgdon, Joshua L. Kihm, Alex Settle, Dirk Grunwald, and Daniel A. Connors. Dynamic run-time architecture techniques for enabling continuous optimization.

In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 211–220, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-019-1. doi: http://doi.acm.org/10.1145/1062261.1062296.

[113] Carlos García Qui nones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 269–279, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-056-6. doi: http://doi.acm.org/10.1145/1065010.1065043.

[114] Jeremy Reimer. Valve goes multicore. `http://arstechnica.com/articles/paedia/cpu/valve-multicore.ars`, November 2006.

[115] Michael F. Ringenburg and Dan Grossman. Atomcaml: first-class atomicity via rollback. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 92–104, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-064-7. doi: http://doi.acm.org/10.1145/1086365.1086378.

[116] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. 2007. Component-Based Lock Allocation. In Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT '07).

[117] P. Rosedale and C. Ondrejka. Enabling player-created online worlds with grid computing and streaming. *Gamasutra*, September 2003.

[118] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages

187–197, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-189-9. doi: http://doi.acm.org/10.1145/1122971.1123001.

[119] K. E. Schauser, D. E. Culler, and T. von Eicken. Compiler-controlled multithreading for lenient parallel languages. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture, Cambridge, MA*, pages 50–72, New York, 1991. Springer-Verlag. URL `citeseer.ist.psu.edu/20326.html`.

[120] Yuan Zhang, Vugranam C. Sreedhar, Weirong Zhu, Vivek Sarkar, and Guang R. Gao. 2008. Minimum Lock Assignment: A Method for Exploiting Concurrency among Critical Sections. In Languages and Compilers for Parallel Computing, Jose Nelson Amaral (Ed.). Lecture Notes In Computer Science, Vol. 5335.

[121] Sandya Mannarswamy, Dhruva R. Chakrabarti, Kaushik Rajan, and Sujoy Saraswati. 2010. Compiler aided selective lock assignment for improving the performance of software transactional memory. In Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '10)

[122] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-710-3. doi: http://doi.acm.org/10.1145/224964.224987.

[123] Karl Sims. Particle animation and rendering using data parallel computation. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 405–413, New York, NY, USA, 1990. ACM Press. ISBN 0-201-50933-4. doi: http://doi.acm.org/10.1145/97879.97923.

[124] Yonghong Song, Spiros Kalogeropulos, and Partha Tirumalai. Design and implementation of a compiler framework for helper threading on multi-core processors. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 99–109, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2429-X. doi: http://dx.doi.org/10.1109/PACT.2005.17.

[125] Tim Sweeney. The next mainstream programming language: a game developer's perspective. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 269–269, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-027-2. doi: http://doi.acm.org/10.1145/1111037.1111061.

[126] J. M. Virendra, F. S. Michael, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M.L. Scott. Lowering the overhead of nonblocking software transactional memory. In *UR CSD-TR 893*. University of Rochester, Computer Science Department., March 2006. URL `http://hdl.handle.net/1802/2538`.

[127] Website. Ioquake. `http://www.icculus.org/quake3/`, 2006.

[128] Waliullah M.M. and Stenstrom P., "Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems". In *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium* 2008.

[129] T. Wiegand, G. J. Sullivan, G. Bjntegaard, and A. Luthra. Overview of the h.264/avc video coding standard. *IEEE Trans. Circuits Syst. Video Technol*, 13 (7):560–576, 2003.

[130] V. Yodaiken. The RTLinux manifesto. In *Proc. of The 5th Linux Expo, Raleigh, NC*, March 1999. URL `citeseer.ist.psu.edu/yodaiken99rtlinux.html`.

[131] Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, "Irrevocable transactions and their applications". In *Proceedings of the International Symposium on Parallelism in Algorithms and Architectures* (SPAA) 2008 Pages 285-296

[132] R. N. Zucker. Relaxed consistency and synchronization in parallel processors. Technical Report TR-92-12-05, University of Washington, 1992. URL `citeseer.ist.psu.edu/zucker92relaxed.html`.